НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
“КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ СІКОРСЬКОГО”
ФАКУЛЬТЕТ ІНФОРМАТИКИ ТА ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ
Кафедра автоматизованих систем обробки інформації і управління

«До захисту допущено»
В.о. завідувача кафедри
_____ __*О.А.Павлов*__
(підпис) (ініціали, прізвище)
“___” _____ 2019 р.

Дипломний проект

на здобуття ступеня бакалавра

з напряму підготовки *6.050103 «Програмна інженерія»*

спеціальність *«Програмне забезпечення систем»*

на тему: *Розробка мови програмування для розподілених обчислень на основі моделі акторів*

Виконав: студент 4 курсу, групи ІП-52

Ждан-Пушкін Антон Володимирович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник *асистент Ісаченко Г.В.* _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Консультант з графічної документації *доц. к.т.н. Ліщук К.І.* _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Рецензент *ст. викладач каф. ОТ Виноградов Ю.М.* _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цьому дипломному проекті немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2019 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *автоматизованих систем обробки інформації і управління*
(повна назва)

Рівень вищої освіти – перший (бакалаврський)

Напрям підготовки (програма професійного спрямування) – *6.050103 «Програмна інженерія» (Програмне забезпечення систем)*

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ *О.А. Павлов*
(підпис) (ініціали, прізвище)
“___”__2019 р.

З А В Д А Н Н Я
НА ДИПЛОМНИЙ ПРОЕКТ СТУДЕНТУ

Ждан-Пушкіна Антона Володимировича
(прізвище, ім’я, по батькові)

1. Тема проекту «Розробка мови програмування для розподілених обчислень на основі моделі акторів»

керівник проекту *Ісаченко Георгій Вадимович, асистент*
(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від “23” квітня 2019 р. №1181-с

2. Термін подання студентом проекту «03» *червня 2019 року*

3. Вихідні дані до проекту

Технічне завдання

4. Зміст пояснювальної записки

1) Аналіз вимог до програмного забезпечення: змістовий опис і аналіз предметної області, огляд наявних аналогів, аналіз вимог до програмного продукту, цілі та задачі розробки.

2) Опис розробленої мови програмування: типи та система типізації, активні об’єкти та їх семантика, пасивні об’єкти, виконання у розподіленому середовищі.

3) Конструювання програмного забезпечення: синтаксичний та лексичний аналізатори, семантичний аналізатор, модуль виконання AST, інструмент для налаштування середовищ.

4) Аналіз якості та тестування програмного забезпечення: аналіз якості, опис процесів тестування, опис контрольних прикладів.

5) Впровадження та супровід програмного забезпечення: розгортання програмного забезпечення, робота з мовою програмування, супровід програмного забезпечення.

5. Перелік графічного матеріалу

1) Схема структурна діяльності роботи програмного забезпечення

2) Схема бізнес-процесу

3) Схема структурна класів програмного забезпечення

6. Консультанти розділів проекту

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання «20» *лютого 2019 року*

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломного проекту	Термін виконання етапів проекту	Примітка
1.	*Вивчення предметної області*	*22.02.2019*	
2.	*Аналіз існуючих технологій та методів розв'язання задачі*	*28.02.2019*	
3.	*Постановка та формалізація задачі*	*1.03.2019*	
4.	*Аналіз та розроблення вимог до програмного забезпечення*	*7.03.2019*	
5.	*Розробка та формалізація синтаксису і семантики мови програмування*	*20.03.2019*	
6.	*Моделювання програмного забезпечення*	*28.03.2019*	
7.	*Розробка архітектури програмного забезпечення*	*05.04.2019*	
8.	*Розробка програмного забезпечення*	*14.04.2019*	
9.	*Оформлення пояснювальної записки*	*20.05.2019*	
10.	*Виконання графічних документів*	*24.05.2019*	
11.	*Подання ДП на попередній захист*	*28.05.2019*	
12.	*Подання ДП рецензенту*	*3.06.2019*	
13.	*Подання ДП на основний захист*	*07.06.2019*	

Студент _____ Ждан-Пушкін А.В.
(підпис)

Керівник проекту _____ Ісаченко Г.В.
(підпис)

№ з/п	Формат	Позначення	Найменування	Кількість листів	Примітка
1	А4		Завдання на дипломний проект	2	
2	А4	КПІ.ІП-5206.045490.01.81	Пояснювальна записка	54	
3	А4	КПІ.ІП-5206.045490.02.91	Технічне завдання	11	
4	А4	КПІ.ІП-5206.045490.03.51	Програма та методика тестування	6	
5	А4	КПІ.ІП-5206.045490.04.33	Керівництво програміста	8	
6	А4	КПІ.ІП-5206.045490.05.13	Опис програми	17	
7	А3	КПІ.ІП-5206.045490.06.99 СС	Схема структурна діяльності роботи програмного забезпечення	1	
8	А3	КПІ.ІП-5206.045490.06.99 СБП	Схема бізнес-процесу	1	
9	А3	КПІ.ІП-5206.045490.06.99 СС	Схема структурна класів програмного забезпечення	1	

Зм.	Арк.	ПІБ	Підп.	Дата	КПІ.ІП-5206.045490.00.90		
Розробн.		Ждан-Пушкін А.В.			Відомість дипломного проекту	Літ.	Лист 1 / Листів 1
Керівн.		Ісаченко Г.В.					
Консульт						КПІ ім.Ігоря Сікорського кафедра АСОІУ гр. ІП-52	
Н/контр.		Ліщук К.І.					
В.о.зав.каф.		Павлов О.А.					

АНОТАЦІЯ

До пояснювальної записки входять 5 розділів, 19 таблиць, 9 рисунків, а також 8 посилань на джерела інформації та додаток з програмним кодом. Загальний обсяг складає 60 сторінок.

Метою дослідження є аналіз методів розподілених обчислень та спрощення способів реалізації таких обчислень. Для досягнення такої мети обрано реалізувати мову програмування на моделі акторів – моделі паралельних обчислень. Крім інтерпретатора для виконання мови також розроблено комплекс інструментів для управління розподіленими середовищами, на базі яких має виконуватися розроблена мова.

У першому розділі наведено основні теоретичні відомості щодо моделі акторів, оглянуто її варіації, сучасні та історичні реалізації, а також на їх основі виведено основні функціональні вимоги до мови програмування.

У другому та третьому розділі наведено опис мови програмування, способів її роботи, а також архітектурні та алгоритмічні деталі роботи комплексу.

У четвертому розділі наведені тестування результату роботи та описані способи тестування комплексу у різних умовах та середовищах, а у п’ятому розділі описано спосіб його розгортання.

Також до пояснювальної записки додаються графічні матеріали із схемою діаграми діяльності, схемою структурною класів програмного забезпечення та схемою використання.

КЛЮЧОВІ СЛОВА: МОДЕЛЬ АКТОРІВ, РОЗПОДІЛЕНІ ОБЧИСЛЕННЯ, ПАРАЛЕЛІЗМ, МОВА ПРОГРАМУВАННЯ, АКТИВНИЙ ОБ’ЄКТ, АСИНХРОННІ ПОВІДОМЛЕННЯ, ГОРИЗОНТАЛЬНЕ МАСШТАБУВАННЯ

ABSTRACT

Explanatory note for the diploma thesis consists of 5 chapters, 19 tables, 9 images, 8 sources of information and single annex with source code. Total size is 60 pages.

The purpose of the study is to analyze different methods of distributed computing and simplify implementation of those. In order to achieve this goal a new programming language is proposed and implemented. This language is based on actor computations model. In addition, some investigation and work is done over a tool to create and manage distributed environments, which will run mentioned programming language.

First chapter describes general theory about actor model and variations of it. Modern industry and academic implementations are also described. All this analysis is used to write down the functional requirements of the language.

Second and third chapters defines a language, along with its syntax and semantics. There is also a description of architecture that is used to implement all the system and algorithms that make it work.

Fourth chapter holds notes about testing of the final product and results of those testing. Fifth chapter describes a way to deploy and set up programming language for real-world use.

In addition, graphic annexes are added to the end of explanatory note. Thay contain schemes of activities, diagram of class structure and business-process diagram.

KEYWORDS: ACTOR MODEL, DISTRIBUTED COMPUTING, PARALLELISM, PROGRAMMING LANGUAGE, ACTIVE OBJECTS, ASYNCHRONOUS MESSAGES, HORIZONTAL SCALING

Пояснювальна записка
до дипломного проекту

на тему: Розробка мови програмування для розподілених обчислень на основі моделі акторів

Київ – 2019 року

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ 10

ВСТУП 11

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 13

1.1 Змістовний опис і аналіз предметної області 13

1.1.1 Поняття моделі акторів 13

1.1.2 Класична модель акторів 14

1.1.3 Модель процесів 15

1.1.4 Active objects (Активні об'єкти) 15

1.2 Огляд наявних аналогів 17

1.2.1 Rosetta 18

1.2.2 Erlang 18

1.2.3 ABCL/1 (Actor-based concurrent language) 18

1.2.4 Akka 19

1.2.5 Orleans 19

1.3 Аналіз вимог до програмного продукту 20

1.3.1 Розроблення функціональних вимог 20

1.4 Цілі та задачі розробки 23

Висновок до розділу 24

2 ОПИС РОЗРОБЛЕНОЇ МОВИ ПРОГРАМУВАННЯ 25

2.1 Типи та система типізації 27

2.2 Активні об'єкти і їх семантика 28

2.3 Пасивні об'єкти 29

2.4 Виконання у розподіленому середовищі 30

Висновок до розділу 32

3 КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 33

3.1 Синтаксичний та лексичний аналізатори 34

3.2 Семантичний аналізатор 34

3.3 Модуль виконання AST 34

3.4 Інструмент для налаштування середовищ 43

Висновок до розділу 45

4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ .46

4.1 Аналіз якості 46

4.2 Опис процесів тестування.....46

4.3 Опис контрольних прикладів 47

4.3.1 Тестування мови програмування у межах одного середовища 47

4.3.2 Тестування роботи розподіленого середовища 53

Висновок до розділу 56

5 ВПРОВАДЖЕННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ.....57

5.1 Розгортання програмного забезпечення.....57

5.2 Робота з мовою програмування 57

5.3 Супровід програмного забезпечення 58

Висновки до розділу 58

ВИСНОВОК 59

ПЕРЕЛІК ПОСИЛАНЬ.....60

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ, СИМВОЛІВ, ОДИНИЦЬ, СКОРОЧЕНЬ І ТЕРМІНІВ

Асинхронна передача повідомлень – передача повідомлень без очікування на відповідь, яка не блокує потік виконання і після якої одразу виконується наступна команда програми

Модель акторів – математична модель конкурентних та паралельних обчислень

Потік виконання – контекст, у якому виконуються одне за одним дії програми, найменша одиниця обробки яка може бути виділена ядром операційної системи.

AST (Abstract Syntax Tree) – позначене орієнтоване дерево, в якому вершини співставлені з операторами мови програмування, а листя – з операндами, універсальний спосіб проміжного представлення інструкцій мови програмування

ВСТУП

Комп’ютеризація та інформатизація стали основними течіями розвитку людства та технологій 21 століття. Починаючи з моменту появи та розповсюдження персональних комп’ютерів, вони проникли у майже всі галузі людської діяльності. Однією з найголовніших проблем яку вирішили комп’ютери стали обмін, зберігання та обробка інформації. Саме робота з нею стала одним з найбільших трендів сучасних комп’ютерних наук та дала поштовх новим або старим розробкам. Нейронні мережі, машинне навчання, індустрія «великих даних» - усі ці галузі стали популярні саме через надзвичайну кількість даних, яке людство почало генерувати – іноді у вигляді змістовних текстів, повідомлень, файлів, а іноді – лише як побічних ефектів від використання програм чи історії дій користувача.

Поява таких галузей пов’язана з тим, що об’єми нових даних постійно зростають, і, що важливіше, зростають швидше, ніж потужності процесорів, фізичних носіїв пам’яті тощо. Несподівана різниця у рості оброблюваної інформації та можливостях її обробки призвела до необхідності шукати нові, ефективніші способи роботи з даними.

Одним з рішень сповільнення темпів розвитку комп’ютерного обладнання стала розробка нових підходів та алгоритмів, орієнтованих на паралельне та розподілене виконання. Основною перевагою такого підходу є його гнучкість, у разі зміни об’ємів дуже просто балансувати ефективність обладнання за допомогою горизонтального масштабування – швидкого вилучення чи додавання нових процесорних одиниць до групи працюючих комп’ютерів. Поява такого способу роботи з даними призвела до необхідності створення нових інструментів, які би допомогли автоматизувати та спростити як написання систем, готових до виконання у паралельних та розподілених середовищах, так і допомогти налаштовувати такі середовища.

Існує кілька різних галузей, які надають та вивчають інструменти та способи роботи у розподілених середовищах. Особливе місце серед них займає модель акторів – математична модель, винайдена пів століття тому та успішно удосконалена та застосована у сьогоднішньому світі. Проте, незважаючи на успішність, простота та гнучкість такої моделі призвели до досить різних її інтерпретацій у різних технологічних рішеннях, кожне з яких має як свої недоліки, так і свої переваги.

Метою даної дипломної роботи є аналіз моделі акторів та створення власної мови програмування, яка втілить сукупність кращих ідеї і нині існуючих рішень та стане зручним та простим інструментом для розробки та виконання систем, орієнтованих та паралельні та розподілені обчислення.

1 АНАЛІЗ ВИМОГ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

1.1 Змістовний опис і аналіз предметної області

Мова програмування – це основний інструмент для розробки програмних додатків. На даний момент існує велика кількість мов програмування, що відрізняються одне від одної різними можливостями, фундаментальними ідеями та галузями застосування. Найчастіше їх класифікують за наступними властивостями:

- за так званим «рівнем» - поділяють на низькі та високі в залежності від їх наближеності до машинного коду;
- за основною парадигмою (наприклад, функціональні чи об’єктно орієнтовані);
- за системою типізації (наприклад, статичні чи динамічні) та її силою, яка характеризую спосіб взаємодії між різними типами.

Важливою частиною кожної мови програмування є ідеї, на основі яких розробляється її дизайн. Наприклад, для багатьох функціональних мов програмування основою виступає лямбда-числення, що було розроблене Алонзо Чорчем (Alonzo Church) у першій половині XX століття. Ідеї об’єктно-орієнтованих мов програмування вперше сформувалися в знайомому нам вигляді к мові програмування Smalltalk, створеній Аланом Кеєм (Alan Key) у 1970-х роках. Приблизно у ті ж роки американський дослідник Карл Хьювіт (Carl Hewitt) запропонував так звану «модель акторів» – концепцію для аналізу та дизайну паралельних обчислень.

1.1.1 Поняття моделі акторів

Модель акторів – це математична модель паралельних обчислень, у якій основним примітивом паралельного виконання поняття «актора». В рамках даної моделі виконання будь-якої програми повинно бути описано набором акторів, які певним чином створюють одне одного, відправляють одне одному

повідомлення та реагують на отримані повідомлення. В ході свого розвитку ця модель отримала декількох основних інтерпретацій, одну з яких і буде розглянуто та реалізовано в рамках даної теми[11].

У моделі акторів усі комунікації у системи будуються за допомогою передачі асинхронних повідомлень від актора до актора. Кожен актор має свою ізольовану пам’ять, яка ніяким чином не доступна для інших. Також, одночасно може оброблюватися лише одне повідомлення, а усі подальші зберігаються до черги цього актора і чекають свого виконання. Така структура була запропонована ще на самому початку розвитку моделі, і вона виявилася дуже зручною для паралельних обчислень – адже наявність ізольованої пам’яті та відсутність необхідності синхронізації дозволяла дуже просто налаштовувати паралельну роботу багатьох акторів. Саме через це основною сферою її використання стали конкурентні та паралельні системи.

Варто зазначити, що зазвичай робота моделі акторів розглядається у середовищі, де неможливі збої, тобто, доставка повідомлень завжди гарантована, хоча і не гарантований порядок або часові затримки у доставці. Однак, орієнтованість моделі на ізольовані та самостійні процеси дозволяє досить просто перейти до розгляду моделі в умовах розподіленого середовища, якщо внести до розгляду певні його особливості, наприклад проблеми загублених повідомлень, синхронізації в умовах ненадійної передачі тощо. Деякі з нижче розглянутих варіантів моделі мають власні способи вирішення таких проблем.

1.1.2 Класична модель акторів

Класичною моделлю акторів називають запропоновану Карлом Хьювітом модель акторів. Пізніше її більш детально доробив та розвинув дослідник Гул Ага (Gul Agha). У рамках такої моделі кожен актор – це набор так званих *«поведінок»*, кожна з яких характеризує дії, які виконаються за отримання

повідомлень. У кожен момент часу актор відображає одну с набору *поведінок*, але може змінити її на іншу внаслідок обробки нового повідомлення.

Основною характеристикою класичної моделі є наявність *«принципу ізольованої обробки»*[1][5]. Згідно цьому принципу, обробка нового повідомлення може розглядатися як атомарна дія, і лише одна з них може виконуватися одночасно. Оскільки кожен актор має свою ізольовану пам’ять, то тільки він сам може змінювати свій стан (тобто, свою поведінку) через передачу повідомлення, а так як його обробка – атомарна і єдина доступна операція, то подальша зміна поведінок можлива лише за допомогою подальших повідомлень.

1.1.3 Модель процесів

Концепція актору як процесу була вперше введена мовою програмування Erlang. Вона є досить схожою до класичної моделі, але використовує інші механізми для забезпечення тих самих ефектів. У цій інтерпретації кожен актор представлених не набором поведінок, що змінюються за заданими правилами, а процесом – набором операцій, що виконуються від початку і до кінця, періодично зупиняючись у очікуванні нового повідомлення[8]. За допомогою конструкцій мов чи бібліотек, що реалізують цю модель, досить часто можна описувати систему у стилі класичної моделі, але це не є обов’язковим.

Даний підхід є дуже зручним з прикладної точки зору, адже актори також моделі можуть бути реалізовані за допомогою процесів чи потоків (threads).

1.1.4 Active objects (Активні об’єкти)

Концепція активних об’єктів розвивалась паралельно з початковими ідеями об’єктно-орієнтованого програмування та моделі акторів, тому у ній досить сильно відобразилися ідеї обох. У її рамках акторами виступають об’єкти, кожен з яких має власний потік і контроль виконання, ізольовану

пам’ять, а комунікація між об’єктами також виконується за допомогою повідомлень[7].

Основною особливістю такої інтерпретації є наявність різних способів роботи з повідомленнями[3]. На відміну від інших варіацій моделей, дозволено очікувати на відповідь від актора, до якого було відправлене повідомлення. Активні об’єкти мають три способи відправлення повідомлень: *past, now* та *future.* Порівняння цих типів наведено у таблиці 3.1.

Таблиця 1.1 – Різновиди повідомлень активних об’єктів

Тип	Очікування відповіді (блокування)	Обробка результату	Опис
past	Ні	Ні	Повідомлення у стилі класичної моделі – асинхронне, без очікування на відповідь. Синхронізація відбувається за допомогою передачі у повідомленні адреси автора виклику та повідомлення-відповіді, що відправляється до нього назад.
now	Так	Так	Нагадує за семантикою виклик функції у імперативних мовах – блокує потік та оброблює відповідь коли вона повертається. З точки зору повідомлень актор-відправник у цей момент чекає на повідомлення одного конкретного типу, а інші тимчасово ігнорує.

Продовження таблиці 1.1

future	Ні	Так	Не блокує потік, але відстежує відповідь на повідомлення. Під час отримання відповіді обробляє результат.

У деяких реалізаціях такої концепції також мають місце так звані *passive object* (пасивні об’єкти). Вони існують лише у рамках активного об’єкту, якому належать, а у разі передачі між різними акторами – повністю копіюються до пам’яті нового актора[5]. Таким чином пам’ять все ще ізольована, що дозволяє уникати проблем синхронізації, станів гонки тощо.

1.2 Огляд наявних аналогів

На сьогодні імплементації моделі акторів існують у двох виглядах – у якості мови, яка диктує правила написання програм згідно моделі, або як бібліотеки для мови програмування загального призначення.

У таблиці 1.2 наведено перелік найхарактерніших реалізацій моделі акторів, а також таблиця з короткими характеристиками кожної.

Таблиця 1.2 – Порівняльна характеристика різних реалізацій

Реалізація	Інтерпретація	Парадигма	Обробка повідомлень
Rosetta	Класична модель	Функціональна	Неперервна
Erlang	Модель процесів	Функціональна	Неперервна
ABCL/1	Активні об’єкти	Імперативна	З блокуванням
Akka	Класична модель	Імперативна	З блокуванням
Orleans	Активні об’єкти	Імперативна	Неперервна

1.2.1 Rosetta

Написана дослідниками моделі акторів мова програмування з сімейства мов Lisp. Використовувалася для демонстрації та для академічних досліджень, а також вплинула на подальший розвиток моделі. На сьогодні майже не використовується, але у своїй простоті є дуже вдалим прикладом класичної моделі. Досить часто зустрічається у наукових роботах за темою моделі акторів або використовується як проста та гнучка база для демонстрації нових ідей моделі.

1.2.2 Erlang

Розроблена компанією Ericsson мова програмування, найуспішніша та найпопулярніша на сьогодні реалізація моделі акторів. Була створена на основі моделі процесів для вирішення реальних та конкретних проблем у сфері телекомунікацій. Орієнтується на роботу у розподіленому середовищі, як і мова, що розробляється у рамках диплому, але має динамічну типізацію, а також більше схиляється до функціонального стилю програмування у своїх конструкціях.

Важливим елементом даної мови є орієнтованість на середовище з помилками, перебоями мережі тощо. Оскільки сфера телекомунікацій є дуже вимогливою до надійності програмного забезпечення та вимагає постійного функціонування, були розроблені специфічні способи боротьби з можливими помилками виконання[8][9]. Крім того, наявні досить цікаві можливості із зміною програми під час виконання – ще один результат необхідності мати безперебійні канали зв’язку.

1.2.3 ABCL/1 (Actor-based concurrent language)

Перша мова програмування побудована на моделі активних об’єктів. Схиляється до імперативного стилю описання систем і реалізовує усі три типи передачі повідомлень, що характерні для активних об’єктів[2]. На відміну від

подальших розробок, ще не має підтримки пасивних об’єктів (хоча й не потребує їх, адже основана на Lisp) і також має динамічну типізацію. Не є популярною, як і Rosetta – виступає демонстрацією концепції. Мала пряме продовження у сімействі мов програмування ABCL.

1.2.4 Akka

Бібліотека мови програмування Scala, сучасна інтерпретація класичної моделі акторів і опису їх через поведінки[10]. Також орієнтується на виконання у розподілених системах, що зазвичай не розглядається у класичній моделі акторів. Досить гнучка і підтримує деякі можливості активних об’єктів. Представник статично типізованої моделі акторів, як і мова, розроблювана у рамках дипломної роботи.

Друга за популярності після Erlang технологія на основі моделі акторів. Широко використовується для написання розподілених додатків у різних сферах. Завдячує цим платформі JVM на якій виконується та простоті введення її у проект.

1.2.5 Orleans

Бібліотека від Microsoft для розробки розподілених високо навантажених систем на мові програмування C#. Є досить схожою на Akka за основними ідеями, але основується скоріше на активних об’єктах. Крім того, орієнтується на платформу-конкурента JVM – Microsoft.net, таким чином вводячи модель акторів у світ enterprise. Статично типізована і підтримує різні варіанти передачі повідомлень, як і ABCL/1.

1.3 Аналіз вимог до програмного продукту

1.3.1 Розроблення функціональних вимог

Оскільки кінцевий продукт орієнтований перш за все на використання технічними спеціалістами та адміністраторами, функціональні вимоги описуються з точки зору характеристик та властивостей результату роботи

Схема варіантів використання наведена на рисунку 1.1.

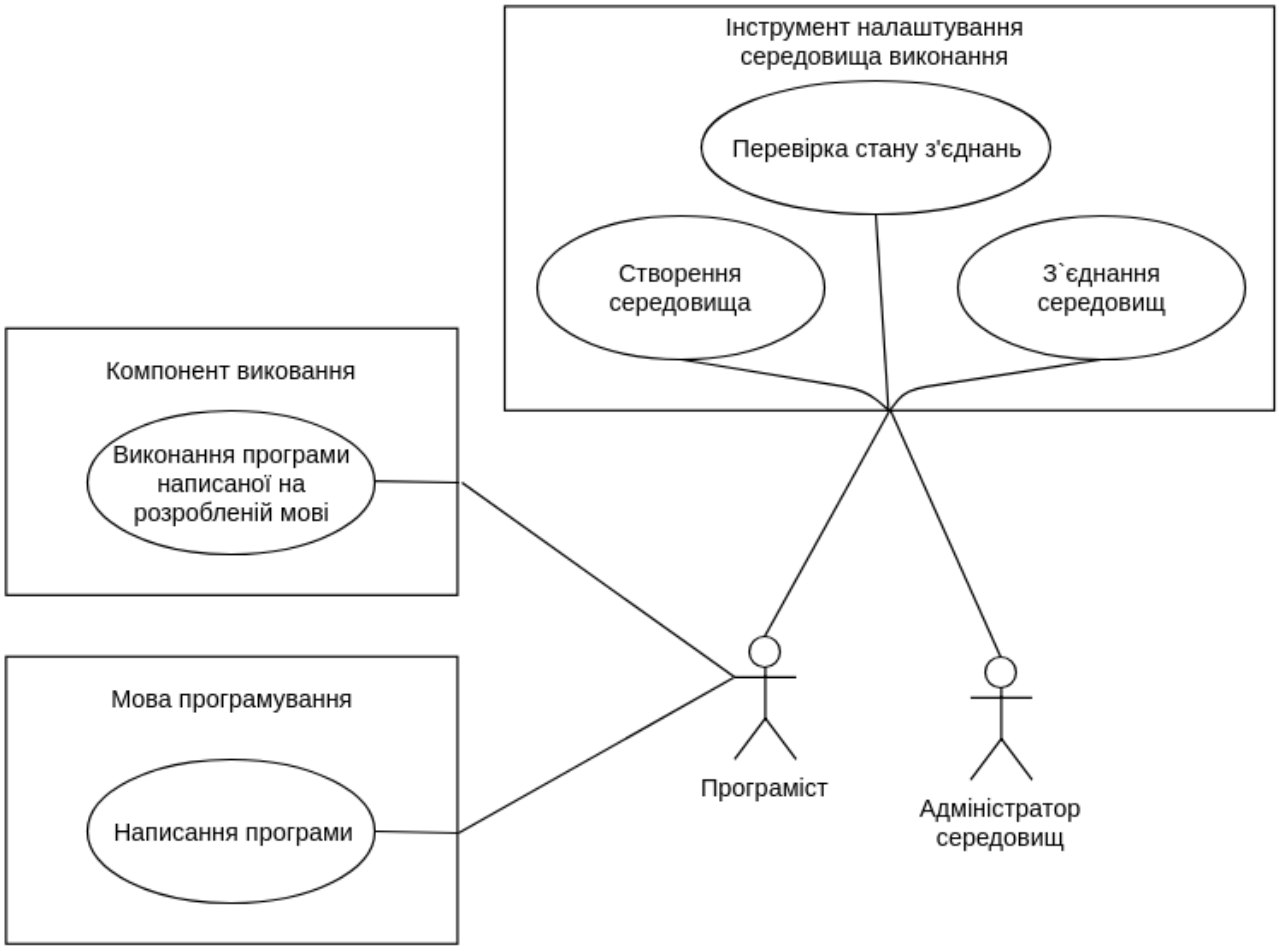


Рисунок 1.1 – Схема варіантів використання

У Таблиці 1.3 наведено функціональні вимоги програмного продукту що мають задовольнити наведені вище варіанти використання. Функціональні вимоги пронумеровано для посилання на них у матриці трасування.

Таблиця 1.3 – Функціональні вимоги

Варіант використання	Функціональна вимога	Пріоритет
1) Написання програми	1.1) Мова має мати описані лексичні та синтаксичні правила, які будуть схожі у використанні до популярних мов	Високий
	1.2) Мова має мати чітко визначену семантику виконання, яка слідує основним ідеям моделі акторів	Високий
	1.3) Розроблена мова має мати чітко описану та надійну систему типів та опиратися на неї у семантиці	Високий
	1.4) Розроблена мова програмування має абстрагуватися від середовища виконання та не залежати від його конфігурації	Високий
	1.5) Має бути наданий базовий комплекс вбудованих засобів для введення та виведення інформації	Високий
	1.6) Мають бути надані засоби декомпозиції задачі, на кшталт об’єктів чи розбиття на модулі	Високий
2)Виконання програми на розробленій мові	2.1) Перед початком виконання виконується перевірка та аналіз тексту програми на відповідність заданій семантиці та лексиці, у разі невідповідності виводиться помилка	Середній

Продовження таблиці 1.3

	2.3) Програма виконується наявності створеного середовища, інакше попереджає про необхідність його створення	Високий
	2.4) Під час виконання програми актори існують у окремих процесах і утилізують усі можливості обладнання серверу	Високий
	2.5) Під час виконання актори не використовують спільну пам’ять і під час передачі даних між собою копіюють їх	Високий
3) Створення середовища	3.1) Компонент контролю середовищ має надавати можливість створювати середовище за файлом конфігурації	Високий
	3.2) Після створення середовища компонент має виводити інформацію про роботу програми у межах середовища та надавати інтерфейс до взаємодії з нею	Низький
4) З’єднання середовищ	4.1) Має існувати можливість створення мережі середовищ за заданим файлом конфігурації з іменованими складовими	Високий
	4.2) Має існувати можливість підключатися до існуючого середовища з анонімного, не вказаного в конфігурації хосту	Високий

Продовження таблиці 1.3

5) Перевірка з’єднання	5.1) Після спроби з’єднання компонент має повідомити про успішність чи неуспішність спроби	Високий
	5.2) У разі успішного з’єднання середовища мають час від часу перевіряти зв’язок одне з одним	Низький

Призначенням розробки є впровадження нової мови програмування та засобів її виконання, які дозволять швидше та простіше у порівнянні з аналогами будувати системи, орієнтовані на виконання у паралельних або розподілених системах.

1.4 Цілі та задачі розробки

Ціль дипломної роботи – створення мови програмування та засобів її виконання для написання програм, які будуть позбавлені найрозповсюдженіших проблем пов’язаних з паралельним виконанням, таких як синхронізація та спільна пам’ять, стани гонки тощо. Також функціонал розроблених інструментів дозволить швидко розгортати написану програму у розподіленому середовищі.

Для досягнення мети розробки необхідно вирішити наступні задачі:

- Розробити дизайн мови програмування згідно моделі акторів;
- У ході розробки дизайну створити просту і гнучку систему типів, що гарантуватиме надійність програм;
- Лексичний аналіз та граматичний аналіз створеної мови, її транслювання у абстрактне синтаксичне дерево;
- Розробити статичний аналізатор для перевірки синтаксичного дерева згідно системи типів та операційної семантики мови програмування;

- Реалізація виконання мови програмування на підтримуваних операційних системах;
- Створення рішення для виконання програм у розподіленому середовищі та конфігурації цього середовища.

Висновок до розділу

В результаті аналізу предметної області було оглянуто основні положення предметного середовища та проаналізовано можливі варіанти моделей акторів, які будуть використовуватися для розробки мови програмування. Під час огляду різновидів моделі акторів біли наголошені їх основні відмінності та проблеми, які вирішуються цими відмінностями.

Також було проведене порівняння конкурентів, а саме актуальних чи концептуальних реалізацій мов програмування та бібліотек на основі моделі акторів. Для кожного з наведених аналогів перелічено їх переваги, які допоможуть у досягненні результату розробки, а також недоліки, які варто врахувати та усунути під час реалізації власної мови програмування.

На основі загальних відомостей на наявних рішень сформовано основна мета розробки та цілі, які необхідно переслідувати для її досягнення. Крім того, описано задачі, що будуть розв’язані для отримання бажаного результату.

2 ОПИС РОЗРОБЛЕНОЇ МОВИ ПРОГРАМУВАННЯ

У рамках даного розділу наведено опис розробленої у рамках даної дипломної роботи мови програмування Frisbee. Опис способу роботи та архітектури мови програмування наведено у розділі 3. Frisbee відноситься до імперативних мов програмування, має просту і гнучку типізацію, дещо схожу на типізацію мови програмування C, а також базові конструкції if та while, ідентичні за синтаксисом до C.

Для попереднього ознайомлення нижче на рисунку 2.1 наведений приклад простої програми на даній мові. З функціональної точки зору програма виконує наступні дії:

а) Відкриття нового сокету для обміну повідомленнями;

б) Отримання першого повідомлення по сокету та його вивід на термінал чи інші засоби виводу;

в) Видалення з тексту повідомлення певної букви;

г) Повернення модифікованого тексту разом із привітанням.

Також для короткого ознайомлення наведемо більш технічний опис інструкцій програми:

Дана програма має наступну структуру:

а) Програма починається з опису імпортів, у даному випадку – імпорту типу LocalSocket із стандартного модулю socket;

б) Після опису імпортованих модулів і типів ідуть декларації типів даного файлу, а саме пасивного типу StringLetterRemover та активного Main. Опис останнього є точкою запуску програми;

```
1  from socket import LocalSocket;
2
3
4  passive StringLetterRemover {
5      String initial;
6
7      def String del(String letter) {
8          [String] letters = this.initial.to_array()
9          val result;
10
11         i = 0;
12         while(i < letters.length()) {
13             if (letters[i] != letter)
14                 result = result + letter;
15         }
16
17         return result;
18     }
19 }
20
21 active Main {
22     def Void run() {
23         Socket sock = spawn LocalSocket("10508");
24
25         String text <= s ! get();
26         io ! print("Got text: " + text);
27
28         text = (new StringLetterRemover(text)).del("a");
29
30         sock ! put("Hello, " + text);
31     }
32 }
```

Рисунок 2.1 – Приклад програми на мові програмування frisbee

в) Кожна з декларацій складається з опису полів об’єктів та методів. Метод run у типу Main є обов’язковим і викликається під час запуску програми;

г) На 7 рядку міститься початок опису методу, у якому наведені тип що повертається та аргументи і їх типи;

д) На рядку 8 оголошується масив із строк та заповнюється результатом виклику метода to_array(). Також відбувається доступ до власного поля за допомогою ключового слова this;

е) 9 рядок демонструє особливість розробленої системи типів – динамічний тип. Таке оголошення змінної дозволяє пропустити для неї перевірки типів, що іноді зручно для моделей, побудованих на акторах;

ж) Приклад циклу та умови наведено на рядках 11 – 15. Також саме операція на рядку 14 над об’єктом result і letter дозволяє визначити, що тип result – строка;

з) 23 рядок знайомить нас із акторами і показує приклад створення активного об’єкту (детальніше про це нижче);

и) Рядки 25, 26 і 30 оголошують відправку повідомлення до активних об’єктів. На відміну від пасивних об’єктів, в них саме такий синтаксис взаємодії;

к) Також на рядку 25 зображено синтаксис для очікування відповіді від іншого активного об’єкту;

л) Вивід програми відбуваєть на рядку 26 за допомогою вбудованого актору (активного об’єкту) io та повідомлення print;

м) 28 рядок – приклад створення пасивного об’єкту та роботи з ним.

2.1 Типи та система типізації

Основою кожної мови програмування є її типи, за допомогою яких виконуються усі операції. У Frisbee наявні чотири базові типи даних:

- Int – цілочисельний тип даних. Операції над цим типом завжди призводять до цілих чисел, у разі ділення відбувається округлення у меншу сторону (таким чином, 5 / 2 = 2);
- String – строковий літерал, підтримує основні операції та має необмежену довжину;
- Bool – логічний тип, має два значення – true та false;
- Void – пустий тип, має єдине значення – void.

Кожен з типів можна оголосити повним або неповним. Повний тип означає, що змінна цього типу завжди має значення, а неповний – що змінна або має значення цього типу, або містить Void. Саме використання неповних типів є єдиним варіантом використання типу Void. Окрім базових типів, є

можливість оголошувати масиви, елементи якого мають певний тип. Масиви з різних типів чи кортежі не імплементовано.

Приклади використання таких типів наведено на рисунку 2.2.

```
# Прості вбудовані типи #
String text = "Hello world!";
Int number = 123;
Bool ready = true;

# Комплексні типи - неповний та масив #
String? name = void;
[Int] ages = [20, 34, 21, 20];

# Масив, що містить пусті значення або інший масив #
[[Bool]?] = [ [true, true], void, [] ];
```

Рисунок 2.2 – Вбудовані типи мови програмування Frisbee

Для розширення базової системи передбачено створення власних типів, які можна поділити на пасивні та активні, що базуються на концепції активних об’єктів, описаних у пункті 1.1.4. У кожного з них є можливість задавати свої поля, тобто кожний з нових типів виступає контейнером для інших значень.

У рамках даної роботи не розглядається детальна взаємодія концепцій ООП та моделі акторів, через що наслідування або інкапсуляція неможливі, а під час створення об’єкту певного власноруч створеного типу необхідно заповнити усі його поля.

2.2 Активні об’єкти і їх семантика

Відносно моделі акторів, у даній реалізації активні об’єкти є акторами, які обмінюються одне з одним повідомленнями за допомогою синтаксису, наведеного у прикладі вище. Актори створюються за допомогою інструкції spawn, після чого з’являються у окремому процесі виконання та є ізольованими від інших процесів[4][6]. Поля актора заповнюються під час його створення і

після цього не є доступними для інших акторів, тобто стають власним ізольованим станом нового об’єкту. Виклик команди spawn повертає посилання, до якого можна відправляти повідомлення, а також яке можна передавати між різними активними об’єктами в якості аргументу.

Окрім відправлення повідомлень до актору також є можливість очікувати на отримання результату від нього, як це було наведено у прикладі вище. З точки зору семантики у цьому разі активний об’єкт, що ініціює виклик, очікує на повідомлення від актору-отримувача, а актор отримувач після завершення відправляє повідомлення назад. Таким чином, концепція комунікації лише за рахунок повідомлень не порушується.

Важливо зазначити, що очікування результату ніяк не співвідноситься із оголошеним результатом обробки повідомлення. Навіть якщо результат методу – void, позивач може очікувати на нього для блокування до отримання результату. І навпаки, позивач має право ігнорувати результат та не очікувати кінця обробки, як і задумано класичною моделлю акторів.

Вартим уваги є особливість роботи ключового слова this для активних об'єктів. Воно використовується для доступу до полів і методів об'єкту і має поведінку пасивного об'єкту. Таким чином, якщо ззовні на актор можна впливати лише через повідомлення, то він сам може викликати методи обробки повідомлень як звичайні функції. Проте, синтаксис відправлення повідомлень також працює, що робить роботу із this гнучкою та зручною.

2.3 Пасивні об’єкти

Основною відмінністю пасивних об’єктів є відсутність власного потоку виконання. Тобто, пасивний об’єкт існує і виконується лише у рамках актора якому належить. Виклик методів відбувається аналогічно звичайним об’єктно-орієнтованим мовам програмування і не підтримують виклик повідомлення. Проте, не є забороненою передача повідомлень із пасивного об’єкта. Тобто, з

точки зору дизайну мови, вони є лише способом декомпозиції програмного коду для зручності і повторного використання.

Ініціалізація таких об’єктів відбувається за допомогою ключового слова new і, як і для активних, вимагає повного опису полів.

Передача пасивних типів між акторами дозволена, але у такому разі відбувається повне копіювання об’єкту із пам’яті одного актору до іншого, що гарантує наявність у нього лише одного процесу-власника. Існують інші способи забезпечення сумісного використання різними активними об’єктами одного пасивного, але їх розгляд знаходиться за межами даної дипломної роботи.

2.4 Виконання у розподіленому середовищі

Другою після розробки мови програмування основною задачею даного проекту є створення можливостей для виконання у декількох середовищах, з’єднаних із собою за допомогою мережі. Існує велика кількість різних способів налаштування таких середовищ та варіантів його способу роботи. Для розробленої мови програмування було вирішено використати систему, основану на залежності між середовищами.

Конфігурація середовищ задається за допомогою мови розмітки YAML та опису наявних середовищ та їх адрес. Приклад такої конфігурації наведено на рисунку 2.3. У такому варіанті наявні три різні сервери з різними адресами та портами підключення. Крім того, для двох останніх наведена залежність від першого середовища, тобто саме середовище master і відповідна програма мають бути запущені перед стартом роботи останніх двох середовищ.

```yaml
master:
  ip: 142.93.236.104
  port: 7777

node1:
  ip: 142.93.128.161
  port: 7778
  connections:
    - master

node2:
  ip: 165.22.192.95
  port: 7777
  connections:
    - master
```

Рисунок 2.3 – Приклад конфігурації середовища

Підключення до залежних середовищ ініціюють програми, що запускаються пізніше. Таким чином користувачів спонукають до написання програм, у яких наявні основні сервери, що управляють іншими, та допоміжними серверами, які власне виконують роботу.

Найпростіший приклад того, як виглядають такі програми наведено на рисунку 2.4 та рисунку 2.5. У такому випадку основна програма запускається та лише очікує на нові підключення. Допоміжні середовища під час ініціалізації підключаються до основного і запам’ятовують основний модуль його. Під час запуску програми у допоміжних середовищах, до основного методу передається посилання на головне середовища і у програми є можливість зв’язатися з ним способом, заданим програмістом.

```
# master.frisbee #

active Main {
   val connections;

   def Void run() {
     io ! print("MASTER STARTED");
     this.connections = [];
   }

   def Void connect(val actor) {
     io ! print("New environment connected!");
     this.connections = this.connections + [actor];
   }
}

```

Рисунок 2.4 – Програма для головного середовища

```
# node.frisbee #

active Main {
   def Void run(val master) {
     io ! print("Node started");
     master ! connect(this);
   }
}


```

Рисунок 2.5 – Програма для допоміжного середовища

Висновок до розділу

У даному розділі було проведено огляд мови програмування, яка пропонується як вирішення проблем, описаних у розділі 1 та буде імплементована у рамках даної дипломної роботи. Було продемонстровано систему типів мови, приклад програми, написаний нею, а також оглянуто спосіб роботи програм у розподіленому середовищі.

3 КОНСТРУЮВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Виконання тексту програми завжди поділяється на кілька основних етапів, наприклад, лексичного та синтаксичного аналізу, семантичного аналізу та перевірки типів зокрема, генерація машинного коду тощо. Специфіка даної дипломної роботи додає до них ще нетипових етапів: взаємодія з середовищем виконання та взаємодія різних середовищ виконання між собою. Діаграма компонентів для розробки такого програмного забезпечення зображена на рисунку 3.1.

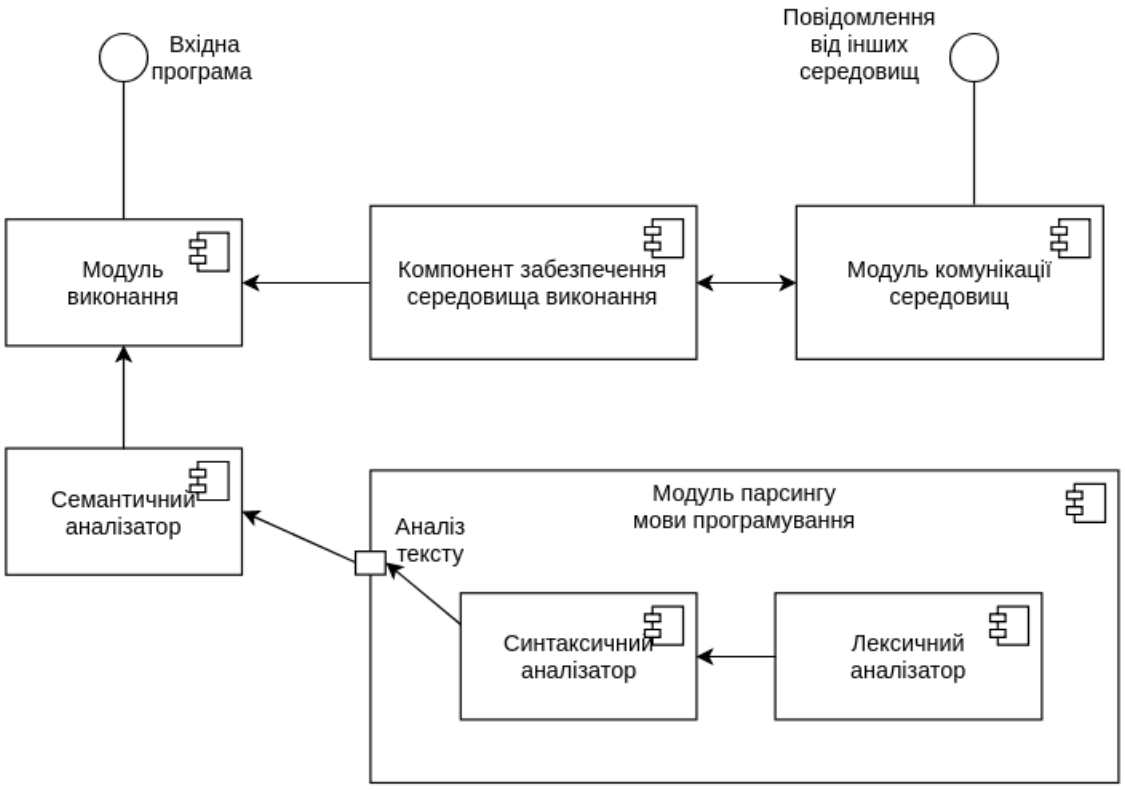


Рисунок 3.1 – Схема компонентів програми

Використання програмного забезпечення кінцевими користувачами вимагає розгортання та користування усіма компонентами наведеними на схемі. Бізнес процеси, що описують взаємодію з розробленим комплектом наведено у графічних матеріалах.

3.1 Синтаксичний та лексичний аналізатори

Дані компоненти виконую роль попереднього аналізу та граматичного розбору лістингу програми. Лексичний аналізатор розбиває текст на лексеми та перевіряє їх коректність, а синтаксичний – за заданими правилами синтаксису будує з лексем абстрактне синтаксичне дерево (далі AST). Саме зі сформованим AST і відбуваються усі подальші дії – аналіз та виведення типів, перевірка семантики, виконання тощо.

Дана частина програмного продукту була розроблена на мові програмування Haskell із застосуванням інструментів Alex та Happy – генераторів високопродуктивних аналізаторів текстів. Компонент представляє собою файл, що можна виконати, та який приймає на вхід текст та видає синтаксичне дерево текстом для подальшої обробки.

3.2 Семантичний аналізатор

Перевірка семантики є важливим етапом, який дозволяє визначити усі можливі помилки програми ще до її виконання. У рамках даного проекту основної семантичної перевіркою є перевірка типів, а саме коректність імпортованих бібліотек, операції над різними типами тощо. Крім того, відбувається велика кількість дрібних перевірок, наприклад, звернення до акторів лише через повідомлення та ізольованість його полів ззовні.

3.3 Модуль виконання AST

Даний компонент відповідає за виконання генерованого та попередньо перевіреного семантично синтаксичного дерева. Оскільки у рамках даної курсової роботи не розглядаються варіанти імплементації виконання моделі акторів, було обрано виконувати код за допомогою інтерпретації. Оскільки парсинг вихідних текстів програм виконано за допомогою мови програмування Python, яку досить важко налаштувати для сумісної з Haskell роботи, також були написані засоби для виклику та зчитування результаті роботи

синтаксичного аналізатору. Усі модулі компоненту виконання програми наведено та описано у таблиці 3.1.

Таблиця 3.1 – опис модулів компоненту виконання

Модуль	Призначення модулю
ast_generator.py	Використовується суто для розробки і необхідний для кодогенерації початкового шаблону модулю ast_def.py. Огляд цього модулю не є необхідним саме за рахунок його виключно допоміжної функції. За своєї роботи він генерує початковий шаблон класів AST мовою програмування Python.
declaration.py	Описує класи AST які відображають оголошення нових типів та методів типів. Також містить базову логіку виконання цих типів
program.py	Має опис основної структури файлу програми на даній мові з точки зору AST
imports.py	Описує класи AST для аналізу імпортованих модулей та імпортованих типів
parser.py	Відповідає за виклик синтаксичного аналізатору та переносу його результатів з тексту до класів мови програмування Python. Таким чином, результатом роботи модулю є необроблене синтаксичне дерево
types.py	Описує класи типів та має основну логіку з їх аналізу та перевірки

Продовження таблиці 3.1

statements.py	Має основну логіку з виконання команд, відправкою повідомлень тощо. Важливим елементом роботи даного модулю є поняття «контексту виконання», який передається між вершинами дерева під час виконання і містить інформацію про процес у якому виконується конкретна дія, область видимості та актуальне значення змінних. Детальний опис класів даного модулю наведено далі у розділі
expressions.py	Містить логіку роботи обчислень виразів мови програмування, таких як оператори, виклик функцій та доступ до полів об’єктів тощо. Детальний опис класів даного модулю наведено далі у розділі
active_object.py	Описує основні дії активних об’єктів, такі як обробка повідомлення, а також забезпечує створення нових акторів у окремих процесах та їх підключення до середовища. Крім того, оголошує «проксуючі актори» - посилання на активні об’єкти, які можна передавати у повідомленнях
passive_object.py	Описує основну логіку роботи пасивних об’єктів та їх методів, таких як доступ до функцій чи поля
builtin_types.py	Реалізує вбудовані види акторів та містить їх логіку на базовій мові програмування, основується на базових класах пасивних та активних об’єктах
global_conf.py	Містить глобальні конфігураційні дані, необхідні для виконання програми, такі як інформація про доступні типи, адреси підключення до середовища тощо

Продовження таблиці 3.1

loader.py	Контролює зчитування файлу та виконання програми. Під час виклику завантажує усі необхідні вихідні файли (включно з імпортованими) , а також знаходить та ініціалізує початковим повідомленням головний клас програми.
connector.py	Містить інтерфейс та реалізацію підключення до середовища та виступає зв’язковим між середовищем і процесами

Найважливішими з точку зору виконання є модулі expressions.py та statements.py, адже вони оголошують логіку основних компонентів будь-якої мови програмування – вирази та команди. Виразами є такі конструкції мови, які можна обрахувати в певних умовах, наприклад, операція над числами або виклик функції. Основною характеристикою виразів є наявність значення, що повертається після обчислення. Команди, в свою чергу, лише контролюють виконання програми і її стан, записуючи чи змінюючи його в залежності від прорахованих виразів. До команд належать команди циклу, розгалуження, присвоювання значення тощо.

Після виконання кожен вираз перетворюється в новий, який називається результуючим. Результуючі вирази є фінальними значеннями і є найменшими неподільними складовими частинами програми. До фінальних виразів належать значення, такі як число, строка, масив чи пусте значення, а також активні та пасивні об’єкти. У рамках реалізації фінальні значення виділені за допомогою класу FinalExp. У таблиці 3.2 наведено основні його методи, а також базові методи класів акторів, пасивних об'єктів та виразів.

Таблиця 3.2 – основні методи класів обчислюваних виразів

Клас	Назва методу	Опис
BaseExp	evaluate	Виконує обчислення виразу. Усі наслідники класу мають перевизначати цей метод для опису власної логіки обчислення. Використовує контекст виконання для доступу до значень змінних
FinalValue	run_method	Викликає певний метод у фінального об’єкту, використовується як пасивними об’єктами, так і звичайними значеннями, наприклад, “123”.to_int()
ExpPassiveObject	get_field	Описує логіку доступу до поля пасивного об’єкту, зчитуючи його із контексту на повертаючи
ExpPassiveObject	set_field	Записує нове значення змінної до поля об’єкту
ExpActiveObject	get_field	Аналогічно пасивному об'єкту, використовується разом з this, коли актор звертається сам до себе
ExpActiveObject	set_field	Аналогічно пасивному об'єкту, використовується разом з this, коли актор звертається сам до себе
ExpActiveObject	proceed_message	Описує логіку обробки нового повідомлення, яке передається разом з параметрами повідомлення

Кожен вид виразу розробленої мови програмування має свій клас, що наслідується від BaseExp та має свою логіку обчислення. У таблиці 3.3

наведено перелік усіх таких класів разом з описом виразів, які вони реалізують, та додаткових полів.

Таблиця 3.3 – Опис класів, що реалізують BaseExp

Назва класу	Опис
ExpOp	Описує логіку роботи бінарних операторів, містить дані про лівий та правий операнд. Основними операторами є +, -, *, /, а також логічні and та or
ExpComOp	Описує логіку роботи бінарних операторів порівняння, яких наявно усього чотири: <, >, !=, ==
ExpArrayGet	Описує логіку отримання за індексом певного елементу з масиву та містить у якості полів інформацію про вираз масиву та вираз для обчислення індексу
ExpArrayValue	Використовується для оголошення масиву та генерується у разі наявності у коді оголошення масиву одразу з елементами
ExpFCall	Реалізує виклик функції у певного пасивного об’єкта, повертає результат роботи цієї функції. Використовується для будь-якого фінального значення
ExpFieldAccess	Реалізує доступ до певного поля пасивного об’єкту або this активного об'єкту
ExpInt	Фінальний вираз, містить значення цілого числа, обчислюється у самого себе

Продовження таблиці 3.3

ExpVoid	Фінальний вираз, містить пусте значення, обчислюється у самого себе
ExpString	Фінальний вираз, містить значення рядку строки, обчислюється у самого себе
ExpBool	Фінальний вираз, містить логічне значення, обчислюється у самого себе
ExpIdent	Реалізує доступ до певної змінної у області видимості, використовує контекст виконання для отримання значення
ExpNewPassive	Описує створення нового активного об’єкту, повертає фінальне значення з ним, крім того ініціалізує його у пам’яті за допомогою аргументів, що передаються разом з new
ExpSpawnActive	Описує створення нового актору, запускає його у окремому процесі та повертає фінальне значення з посиланням на нього, крім того ініціалізує його поля та пасивний об’єкт для доступу актора досамого себе
ExpThis	Посилання на this, повертає пасивний об’єкт з можливістю доступу до полів та методів, що є власником методу який на даний момент виконується
ExpNot	Реалізує унарний оператор логічного заперечення та повертає інвертоване логічне фінальне значення

Продовження таблиці 3.3

ExpIO	Фінальний вираз, містить значення цілого числа, обчислюється у самого себе
ExpArray	Фінальний вираз, містить масив із певними значеннями, обчислюється у самого себе
ExpExp	Вираз, взятий у дужки, необхідний для коректного задання послідовності операцій
ExpPassiveObject	Фінальний вираз, містить значення пасивного об’єкту, його полів у опис методів, які можна викликати, разом із їх логікою
ActiveProxy	Фінальний вираз, посилання на актор, яке повертається під час обчислення виразу створення нового актору

Команди даної мови програмування реалізовано схожим чином – вони усі наслідуються від BaseStatement та реалізують його єдиний метод, який відповідає за контроль стану програми. Під час виконання складовими кожної команди є вирази та інші команди. Детальний опис наявних класів команд та їх значення наведено у таблиці 3.4.

Таблиця 3.4 – Опис класів, що реалізують BaseStatement

Назва класу	Опис
SList	Використовується у разі оголошення блоку команд. Основним прикладом використання є тіло циклу чи умовного оператору – вони можуть бути як одною командою, так і блоком із декількох команд

Продовження таблиці 3.4

SIfElse	Команда умовного оператору, містить дані про різні гілки виконання та вираз умови, і виконує різні гілки в залежності від результату виразу
SWhile	Описує цикл, містить інформації про команди його тіла, які можуть бути або блоком команд, або однією командою, а також вираз, який обчислюється кожної ітерації
SReturn	Команда повернення результату з функції, змушує припинити виконання та моментально повернути результат до викликача
SEqual	Записує значення до певної змінної, перед цим обчислює значення, щоби до області видимості попадали тільки фінальні значення
SEqualField	Записує значення до певного поля об’єкту, записуване значення завжди є фінальним об’єктом
SVarDeclEqual	Оголошення нової змінної та одразу ініціалізація її новим значеннями, також використовується для семантичного аналізу та перевірки типів
SVarDecl	Оголошення нової змінної, але без ініціалізації її, використовується виключно для перевірки типів та семантичного аналізу
SArrayEqual	Запис фінального виразу до певного порядкового елементу масиву

Продовження таблиці 3.4

SSendMessage	Відправлення асинхронного повідомлення до іншого активного об’єкту, без очікування на відповідь
SWaitMessage	Відправлення повідомлення до іншого активного об’єкту, але очікування на відповідь, після отримання відповіді запис її до певної змінної
SExp	Просто вираз, ніяк не впливає на значення контексту виконання

3.4 Інструмент для налаштування середовищ

Окрім виконання вихідного тексту програми велика частина усієї логіки програмного комплексу припадає на інструмент для створення, налаштування та з’єднання середовищ. У розділі 2 «Опис розробленої мови програмування» було наведено спосіб з’єднання та конфігурації розподілених середовищ, тому у цьому розділі акцент зроблено на алгоритми роботи цього модулю.

Під час запуску інструменту на його вхід подається конфігурація, яка описує адресу та порт, на якому середовище приймає нові підключення. Крім того, середовище створює черги, у які будуть складатися повідомлення для відповідних акторів. Під час підключення нової програми до середовища, воно віддає модулю виконання програми інформацію про такі черги і починає вести запис про усі актори даного середовища.

Якщо для запущеного середовища необхідно виконати під’єднання до інших середовищ, то воно створює зв’язок із сокетом відповідного залежного середовища та отримує від нього інформацію про основні актори, яку потім передає до модулю виконання для ініціалізації зв’язку. Детальніше цей алгоритм зображений на рисунку 3.2.

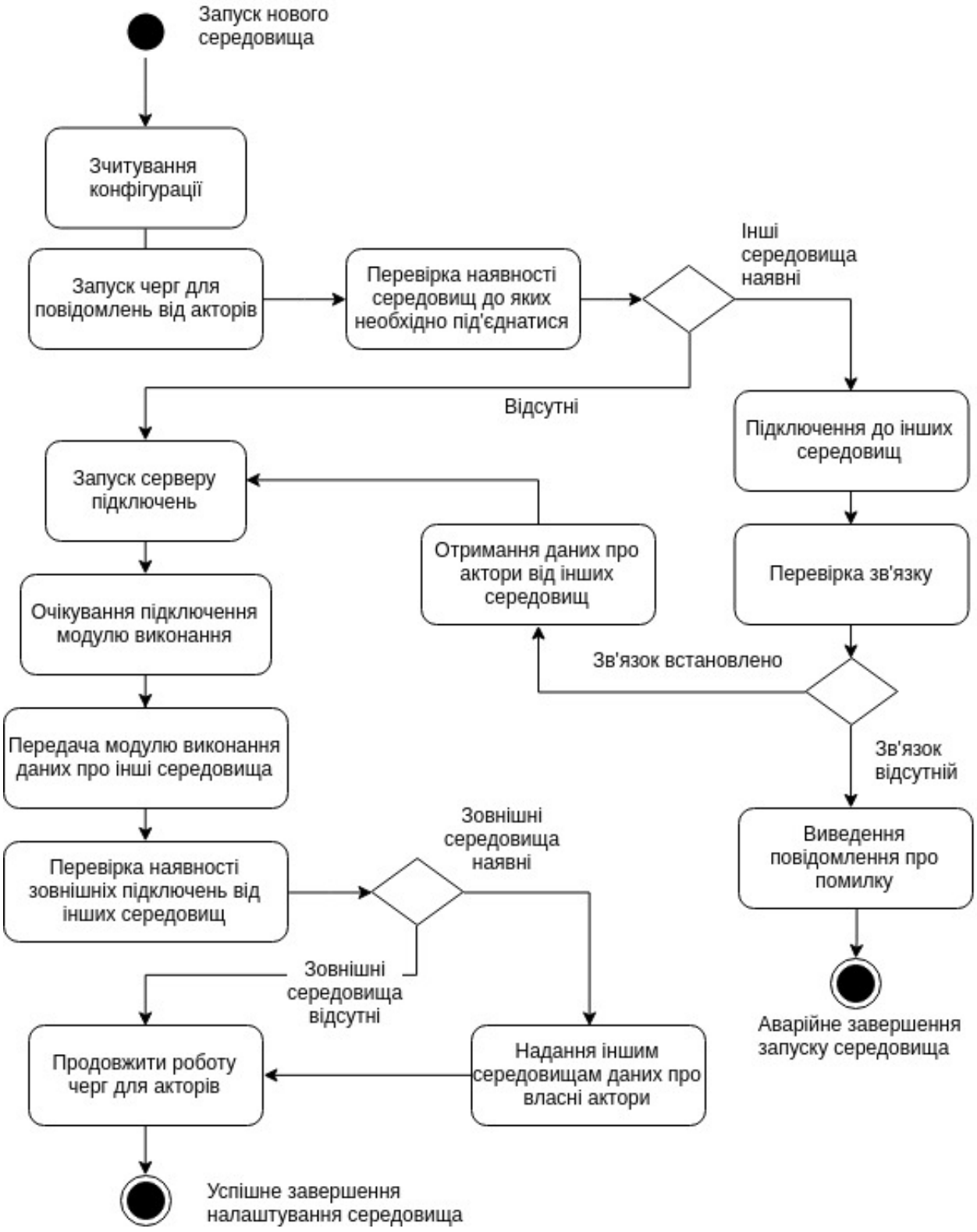


Рисунок 3.2 – Схема діяльності інструменту роботи з середовищами

Висновок до розділу

У рамках даного розділу було оглянуто архітектурну схему роботи розробленого комплексу програм. Також було описано, як саме і за допомогою яких технологій функціонують кожна складова частина цього комплексу. Для модулю виконання програми було наведено детальний опис основних методів та класів, які інкапсулюють логіку виконання тексту програми. Щодо інструменту для поєднання середовищ, було наведено діаграму діяльності даного компоненту, який ілюструє принцип роботи різних середовищ.

4 АНАЛІЗ ЯКОСТІ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Аналіз якості

Для оцінювання отриманого програмного продукту та перевірки його на відповідність наведеним вимогам наявний етап його тестування. Такі заходи необхідні для гарантування коректності розробленого проекту та його надійності.

У ході тестування перевірці підлягають усі компоненти програмного продукту. Крім того, оглядається як відповідність вимогам, наведеним у розділі 1.3, так і відповідність до опису мови програмування, наведеному у розділі 2, а саме наведеним синтаксичним та семантичним правилам.

4.2 Опис процесів тестування

Для перевірки відповідності функціональним вимогам проведена черга контрольних тестів, які перевіряють певні функціональні вимоги. У рамках кожної функціональної вимоги проводиться низка контролів, які дозволяють також гарантувати відповідність семантиці розробленої мови програмування.

У якості вхідних даних для контролю виконання використовуються програми, написані за допомогою розробленої мови програмування. Згідно зі специфікацією для кожного вхідного тексту виводиться очікуваний у разі коректної роботи результат. Також у розділі 4.3.2 проведено тестування виконання програми на розподілених середовищах та власне підключення різних середовищ і тестування його. У цих розділах вхідними даними також виступають конфігурації мереж.

Результати процесу перевірки якості представлені за допомогою матриці трасування, наведеної у таблиці 4.1.

Таблиця 4.1 – Матриця трасування

Функціональна вимога	Ідентифікатор контрольного прикладу					
	SP1 – SP5	SP5 – SP9	SE1	DE1	DE2	DE3
Аналіз програми до виконання	X	X				
Виконання програми на розробленій мові		X		X		
Створення середовища			X	X		
З’єднання середовищ				X	X	X
Перевірка з’єднання					X	X

4.3 Опис контрольних прикладів

4.3.1 Тестування мови програмування у межах одного середовища

У рамках даного пункту усі контрольні перевірки виконуються в однакових умовах: створене одне середовище, вихідний текст програми подається та інтерпретатор як вхідна дія. Секція «початковий стан» для даного розділу опускається, адже є ідентичною для усіх тестів – початкове середовище створене та успішно працює.

Таблиця 4.1 – Обробка лексичної помилки

Ідентифікатор	SP1
Мета тесту	Перевірка обробки лексичної помилки (одинарні лапки замість подвійних)

Вхідна програма	`active Main {`
	`  def Void run() {`
	`  io ! print(‘Hello world’);`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Виведення повідомлення про помилку
Фактичний результат	Програма не завершилася успішно, виведена помилка: `>>> [error] Lexical error at line:3`

Таблиця 4.2 – Обробка синтаксичної помилки

Ідентифікатор	SP2
Мета тесту	Перевірка обробки синтаксичної помилки (відсутність типу методу)

Вхідна програма	`active Main {`
	`  def run() {`
	`  io ! print("Hello world");`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Виведення повідомлення про помилку
Фактичний результат	Програма не завершилася успішно, виведена помилка: `>>> [error] Method declaration missing return type`

Таблиця 4.3 – Можливості динамічної типізації

Ідентифікатор	SP3
Мета тесту	Перевірка відсутності синтаксичного аналізу для анонімних типів

Вхідна програма	`active Main {`
	`  def Void run() {`
	`  val s = 123;`
	`  io ! print(s);`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Успішний вивід повідомлення
Фактичний результат	Програма не завершилася успішно, виведена помилка: `>>> [error] Printing not string is prohibited`

Таблиця 4.4 – Взаємодія з порожнім значенням

Ідентифікатор	SP4
Мета тесту	Перевірка неможливості повного типу мати значення void

Вхідна програма	`active Main {`
	`  def Void run() {`
	`  String x = void;`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Виведення повідомлення про помилку
Фактичний результат	Програма не завершилася успішно, виведена помилка: `>>> [error] Type String cannot store void`

Таблиця 4.5 – Перевірка ізольованості полів акторів

Ідентифікатор	SP5
Мета тесту	Перевірка ізольованості полів активних класів

Вхідна програма	`active Example {`
	`  val x;`
	`}`
	
	`active Main {`
	`  def Void run() {`
	`  val test = spawn Example("PRIVATE");`
	`  io ! print(test.x);`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Виведення повідомлення про помилку
Фактичний результат	Програма не завершилася успішно, виведена помилка: `>>> [error] Cannot access attribute ‘x’ of actor`

Таблиця 4.6 – Передача повідомлення від актора до актора

Ідентифікатор	SP6
Мета тесту	Перевірка передачі повідомлення іншому актору

Вхідна програма	`active Example {`
	`  def Void say_hello(val to) {`
	`  io ! print("Hello to " + to);`
	`  }`
	
	`}`
	
	`active Main {`
	`  def Void run() {`
	`  val actor = spawn Example();`
	`  actor ! say_hello("Anton");`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Успішне виконання програми та вивід тексту
Фактичний результат	Успішний вивід повідомлення: `>>> [out] Hello to Anton`

Таблиця 4.7 – Передача повідомлення до актора і очікування відповіді

Ідентифікатор	SP7
Мета тесту	Перевірка передачі повідомлення разом з очікуванням відповіді

Вхідна програма	`active Example {`
	
	`  def Int add_one(val counter) {`
	`  return 1 + counter;`
	`  }`
	
	`}`
	
	`active Main {`
	
	`  def Void run() {`
	`  val actor = spawn Example();`
	`  val two <= actor ! add_one(1);`
	
	`  io ! print(two.to_string());`
	`  }`
	
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Успішне виконання програми, виведення оновленого значення
Фактичний результат	Успішний вивід повідомлення: `>>> [out] 2`

Таблиця 4.8 – Копіювання пасивних об’єктів під час передачі

Ідентифікатор	SP8
Мета тесту	Перевірка повного копіювання пасивних об’єктів у разі передачі

Вхідна програма	`passive PassObj {val x;}`
	
	`active Example {`
	`  def PassObj set_new(PassObj obj) {`
	`  obj.x = "new value";`
	`  return obj.x`
	`  }`
	`}`
	
	`active Main {`
	`  def Void run() {`
	`  val old_obj = new PassObj("initial");`
	`  val actor = spawn Example()`
	
	`  val new_obj <= actor ! set_new(old_obj);`
	
	`  io ! print("Old value is " + old_obj);`
	`  io ! print("New value is " + new_obj);`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Успішне виконання програми, виведення двох різних значень: перше незмінне, а отримане у відповідь - оновлене

Фактичний результат	Успішний вивід повідомлення:
	`>>> [out] Old value is initial`
	`>>> [out] New value is new value`

Таблиця 4.9 – Передача посилання на актор як аргумента

Ідентифікатор	SP9
Мета тесту	Перевірка можливості передачі активного об’єкту за допомогою повідомлення та ключового слова this

Вхідна програма	`active Mirror {`
	`  def Void call_back(val caller) {`
	`  caller ! print("Mirror");`
	`  }`
	`}`
	
	`active Main {`
	`  def Void print(val from) {`
	`  io ! print("Got call from " + from);`
	`  }`
	
	`  def Void run() {`
	`  val mirror = spawn Mirror();`
	`  mirror ! call_back(this);`
	`  }`
	`}`

Виконання тесту	Запуск програми
Очікуваний результат	Успішне виконання програми, вивід повідомлення з іменем класу - викликача
Фактичний результат	Успішний вивід повідомлення: `>>> [out] Got call from Mirror`

4.3.2 Тестування роботи розподіленого середовища

Нижче наведено контрольні приклади, які перевіряють вимоги щодо створення, з’єднання та сумісної роботи кількох середовищ. У якості вхідних даних виступають конфігураційні файли середовищ. Для тестування приймемо існування комп’ютерної мережі з адресою та маскою 192.168.0.0/29 (містить адреси від 192.168.0.1 до 192.168.0.6).

Таблиця 4.10 – Помилка виконання без середовища

Ідентифікатор	SE1
Мета тесту	Перевірка помилки під час спроби запуску програми без підключеного середовища
Передумови	Відсутні будь-які створені середовища, програма запускається із певною конфігурацією

Вхідні дані	Конфігурація:
	`master:`
	`  ip: 192.168.0.1`
	`  port: 7778`

Проведення тесту	Запуск середовища на основному сервері
Очікуваний результат	Успішне виконання програми, вивід повідомлення про неможливість знайти середовище
Фактичний результат	Середовища під'єднані, виведені повідомлення: `>>> [error] Cannot detect running `master` env`

Таблиця 4.11 – З'єднання кількох середовищ

Ідентифікатор	DE1
Мета тесту	Перевірка успішного з'єднання кількох середовищ під час їх запуску
Передумови	Мережа створена та зв'язок між окремими її членами стабільний і наявний, головні середовище та програма запущені

Вхідні дані	Сумісна конфігурація:
	`master:`
	`  ip: 192.168.0.1`
	`  port: 7778`
	`node1:`
	`  ip: 192.168.0.2`
	`  port: 7778`
	`node2:`
	`  ip: 192.168.0.4`
	`  port: 7779`

Продовження таблиці 4.11

Проведення тесту	Запуск середовища на основному сервері, потім – на допоміжних серверах, перегляд інформації з основного середовища
Очікуваний результат	Успішне виконання програми, вивід повідомлення з іменем класу - викликача

Фактичний результат	Середовища під’єднані, виведені повідомлення:
	`>>> [node1] Connected to master`
	`>>> [node2] Connected to master`

Таблиця 4.12 – Помилка підключення до головного середовища

Ідентифікатор	DE2
Мета тесту	Перевірка помилки під час спроби підключення допоміжного серверу до головного без запущеного головного серверу
Передумови	Мережа створена та зв’язок між окремими її членами стабільний і наявний, але основне середовище не створене

Вхідні дані	Сумісна конфігурація:
	`master:`
	`  ip: 192.168.0.1`
	`  port: 7778`
	`node:`
	`  ip: 192.168.0.2`
	`  port: 7778`

Проведення тесту	Запуск середовища на допоміжному сервері
Очікуваний результат	Вивід повідомлення про помилку
Фактичний результат	Середовище не під’єднане, виведені повідомлення: `>>> [error] Cannot detect running master env`

Таблиця 4.13 – Помилка підключення до середовища без програми

Ідентифікатор	DE3
Мета тесту	Перевірка помилки під час спроби підключення допоміжного серверу до головного без запущеної програми на ньому
Передумови	Мережа створена та зв’язок між окремими її членами стабільний і наявний, основне середовище запущено, але програма не виконується

Вхідні дані	Сумісна конфігурація:
	`master:`
	`  ip: 192.168.0.1`
	`  port: 7778`
	`node:`
	`  ip: 192.168.0.2`
	`  port: 7778`

Проведення тесту	Запуск середовища на основному сервері, потім – на допоміжному
Очікуваний результат	Вивід повідомлення про помилку
Фактичний результат	Середовища під’єднані, виведені повідомлення: `>>> [error] Program is not running on `master` env`

Висновок до розділу

У даному розділі було проведено ряд перевірок розробленого інтерпретатору та інструментарію для з’єднання середовищ. За результатами тестування можна сказати, що розроблений продукт відповідає функціональним вимогам та описаній семантиці та синтаксису мови програмування.

5 ВПРОВАДЖЕННЯ ТА СУПРОВІД ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

5.1 Розгортання програмного забезпечення

Розгортання програмного забезпечення для запуску програм, написаних розробленою мовою програмування, складається з розгортання інтерпретатора розробленої мови та створення та налаштування середовищ, у якому будуть виконуватися програми. Варто зазначити, що тестування і розробка велися з орієнтацією та операційні системи *nix, і хоча використовувані засоби підтримують інші платформи, коректність роботи в них не гарантуються.

Середовище використовує мову програмування python та бібліотеку ZeroMQ для своєї роботи. Необхідні для встановлення бібліотеки вказані у файлі requirements.txt та можуть бути встановлені за допомогою пакетного менеджеру pip. Також під час розгортання необхідно вказати конфігураційний файл, що буде використовуватися для середовища.

Компонент лексичного та синтаксичного аналізу вимагає встановлення Glasgow Haskell Compiler для компіляції у виконавчий файл, а також інструменту компонування додаткових бібліотек, наприклад, stack build tool або cabal, які можуть бути встановлені з офіційних репозиторіїв цільової операційної системи. Після компіляції результуючий виконавчий файл має бути скопійовано у директорію, що містить модуль виконання програм.

Компонент семантичного аналізу та виконання програми використовує мову програмування Python3 та пакетний менеджер pip, за допомогою якого необхідно встановити бібліотеки, наведені у файлі requirements.txt.

5.2 Робота з мовою програмування

Опис синтаксису та семантики мови програмування наведено у розділі 2 «Опис розробленої мови програмування». Детальну інструкцію з запуску і використання наведено у додатку «Інструкція користувача».

5.3 Супровід програмного забезпечення

Супровід програмного забезпечення перш за все передбачає слідкування за технічною інформацією, яку виводять запущені середовища, та за станом мережі, адже дані програмні засоби не пристосовані до роботи у нестабільному з’єднанні.

Висновки до розділу

У розділі було розглянуто необхідні інструменти та сторонні бібліотеки, які використовуються для розгортання розробленого програмного забезпечення та його запуску. Також було оглянуто необхідні заходи для супровіду програми під час виконання та розроблено інструкцію, поміщену у окремий додаток.

ВИСНОВОК

Протягом роботи над даним дипломним проектом було розглянуто один з основних способів вирішення проблеми спрощення горизонтального масштабування за допомогою введення моделі акторів. Був проведений ґрунтовний аналіз предметної області та оглянуто вже існуючі технічні рішення та програмні продукти. На основі результату цих досліджень було розроблено функціональні вимоги до бажаного програмного забезпечення та описано мову програмування, яка гарантує забезпечення цих вимог.

Для описаної мови було прораховано архітектуру. На основі реалізованої архітектури були прийняті рішення щодо технологій, які слід використовувати для реалізації її. Розроблений інтерпретатор та інструменти для виконання у розподіленому середовищі були протестовані на відповідність вимогам та продемонстровано у використанні у інструкції користувача. Результат роботи показує можливості використання даної мови для написання реальних програм за забезпечує простіше та зручніше використання кількох серверів.

У розділі «Впровадження та супровід програмного забезпечення» було наведено основні вимоги до запуску продукту та описано спосіб його розгортання для написання розподілених програм.

Перспективою розвитку даної мови програмування та інструментів для неї перш за все мають стати розширення системи типізації, а також прискорення швидкодії. Систему типізації має сенс розширювати для відповідності сучасним об’єктно-орієнтованим мовам програмування для забезпечення кращої надійності, а швидкодія необхідна для кращої ефективності використання кожного окремого середовища. Також має сенс розглянути введення інших типів повідомлень між акторами, за аналогією з іншими сучасними мовами.

ПЕРЕЛІК ПОСИЛАНЬ

1) History of the actor model [Електронний ресурс] // Wikipedia – 2019 – https://en.wikipedia.org/wiki/History_of_the_Actor_model

2) Carl Hewitt, Henry Baker. “Laws for Communicating Parallel Processes” // Massachusetts Institute of Technology, 1977

3) Joeri De Koster, Tom Van Cutsem, Wolfgang De Meuter. “43 Years of Actors: A Taxonomy of Actor Models and Their Key Properties” // Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control (AGERE 2016), 2016

4) Joe Armstrong. “Making reliable distributed systems in the presence of software errors” // Royal Institute of Technology, 2003

5) Denis Caromel, Ludovic Henrio, Bernard Paul Serpette. “Asynchronous and Deterministic Objects” // Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL 04), 2004

6) Thomas Rouvinez. “Comparison of Active Objects and the Actor Model” // University of Freiburg, 2014

7) Erlang Programming Language [Електронний ресурс] // Erlang – 2019 – https://www.erlang.org/docs

8) Akinori Yonezawa, Jean-Pierre Briot, Etsuya Shibayam “Object-Oriented concurrent programming in ABCL/1” // Department of Information Science Tokyo Institute of Technology Ookayama – 1986

9) Akka Documentation [Електронний ресурс] // Akka – 2019 – https://akka.io/docs

10) R. Greg Lavender, Douglas C. Schmidt “Active object – an Object Behavioral Pattern for Concurrent Programming” // Washington University, 1996

11) Denis Caromel, Eric Madelaine, Ludovic Henrio “Active Objects and Distributed Components: Theory and Implementation” // Formal Methods for Components and Objects, 6th International Symposium (FMCO 2007), 2007

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ О.А. Павлов

“___” _____ 2019 р.

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ НА ОСНОВІ МОДЕЛІ АКТОРІВ

Технічне завдання

КПІ.ІП-5206.045490.02.91

“ПОГОДЖЕНО”

Керівник проекту:

_____ Г.В. Ісаченко

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ А.В. Ждан-Пушкін

Київ – 2019 року

ЗМІСТ

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ....3

2 ПІДСТАВА ДЛЯ РОЗРОБКИ....4

3 ПРИЗНАЧЕННЯ РОЗРОБКИ....5

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ....6

4.1 Вимоги до функціональних характеристик....6

4.2 Вимоги до надійності....6

4.3 Умови експлуатації....6

4.4 Вимоги до складу і параметрів технічних засобів....7

4.5 Вимоги до інформаційної та програмної сумісності....7

4.6 Вимоги до маркування та пакування....7

4.7 Вимоги до транспортування та зберігання....7

4.8 Спеціальні вимоги....7

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ....8

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ....9

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ....11

1 НАЙМЕНУВАННЯ ТА ГАЛУЗЬ ЗАСТОСУВАННЯ

Назва розробки: Розробка мови програмування для розподілених обчислень на основі моделі акторів

Галузь застосування: написання програмістами систем та інструментів для роботи з розподіленими середовищами

Наведене технічне завдання поширюється на розробку мови програмування та системи для її виконання КПІ.ІП-5206.045490, котра використовується для написання надійних програм із можливістю простого горизонтального масштабування та призначена для використання під час розробки систем, що працюють у розподіленому середовищі.

2 ПІДСТАВА ДЛЯ РОЗРОБКИ

Підставою для розробки мови програмування для розподілених обчислень на основі моделі акторів є завдання на дипломне проектування, затверджене кафедрою автоматизованих систем обробки інформації і управління Національного технічного університету України «Київський політехнічний інститут імені Ігоря Сікорського» (КПІ ім.Ігоря Сікорського).

3 ПРИЗНАЧЕННЯ РОЗРОБКИ

Розробка призначена для написання нових програм, систем та інструментів, які мають можливість швидкого горизонтального масштабування та виконання у розподіленому середовищі.

Метою розробки є підвищення ефективності розробки таких програм у порівнянні з традиційними інструментами, збільшення надійності розроблюваних програм відносно існуючих аналогів, спрощення порогу входу до нового способу опису систем, а також спрощення процесу утворення та налаштування розподілених середовищ.

4 ВИМОГИ ДО ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

4.1 Вимоги до функціональних характеристик

4.1.1 Програмне забезпечення повинно забезпечувати виконання наступних основних функції:

4.1.1.1 Для користувача:

- опис та правила виконання розроблюваної мови програмування;
- інструмент для виконання програми написаній на розроблюваній мові програмування;
- інструмент для створення та налаштування розподілених середовищ;
- мати простий та лаконічний спосіб опису конфігурації середовищ.

4.1.2 Розробку виконати на платформі Linux.

4.2 Вимоги до надійності

4.2.1 Передбачити контроль введення інформації.

4.2.2 Передбачити захист від некоректних дій користувача.

4.2.3 Забезпечити перевірку стабільності середовища виконання.

4.3 Умови експлуатації

Не висуваються.

4.4 Вимоги до складу і параметрів технічних засобів

4.4.1 Програмне забезпечення повинно функціонувати на IBM-сумісних персональних комп‘ютерах.

4.4.2 Наявність вільної пам’яті для встановлення програми та збереження результатів.

4.5 Вимоги до інформаційної та програмної сумісності

4.5.1 Програмне забезпечення повинно працювати під управлінням операційних систем сімейства Linux.

4.5.2 Програмне забезпечення повинно бути авторизованим для використання низькорівневих засобів операційної системи.

4.6 Вимоги до маркування та пакування

Вимоги до маркування та пакування не пред‘являються.

4.7 Вимоги до транспортування та зберігання

Вимоги до транспортування та зберігання не пред'являються.

4.8 Спеціальні вимоги

Згенерувати установчу версію програмного забезпечення.

5 ВИМОГИ ДО ПРОГРАМНОЇ ДОКУМЕНТАЦІЇ

5.1 Програмні модулі, котрі розробляються, повинні бути задокументовані, тобто тексти програм повинні містити всі необхідні коментарі.

5.2 Програмне забезпечення повинно мати довідникову систему

5.3 У склад супроводжувальної документації повинні входити наступні документи:

5.3.1 Пояснювальна записка не менше ніж на 50 аркушах формату А4 (без додатків 5.3.2 - 5.3.6).

5.3.2 Технічне завдання.

5.3.3 Керівництво користувача.

5.3.4 Програма та методика тестування

5.4 Графічна частина повинна бути виконана на аркуші формату А3 та А4, котрі включаються у якості додатків до пояснювальної записки:

5.4.1 Схема структурна діяльності процесу роботи програмного забезпечення.

5.4.2 Схема бізнес-процесу роботи програмного забезпечення.

5.4.3 Схема структурна класів програмного забезпечення

6 СТАДІЇ І ЕТАПИ РОЗРОБКИ

№	Назва етапу	Строк	Звітність
1.	Вивчення предметної області	22.02.2019	
2.	Аналіз існуючих технологій та методів розв’язання задачі	28.02.2019	
3.	Постановка та формалізація задачі	1.03.2019	
4.	Аналіз та розроблення вимог до програмного забезпечення	7.03.2019	Специфікації програмного забезпечення
5.	Розробка та формалізація синтаксису і семантики мови програмування	20.03.2019	Специфікації розробленої мови програмування
6.	Моделювання програмного забезпечення	28.03.2019	Схема бізнес-процесу роботи програмного забезпечення
7.	Розробка архітектури програмного забезпечення	05.04.2019	Схема архітектурних компонентів
8.	Розробка програмного забезпечення	14.04.2019	Тексти програмного забезпечення

9.	Оформлення пояснювальної записки	20.05.2019	Пояснювальна записка
10.	Виконання графічних документів	24.05.2019	Графічний матеріал проекту

7 ПОРЯДОК КОНТРОЛЮ ТА ПРИЙМАННЯ

7.1 Види випробувань

Тестування розробленого програмного продукту виконується відповідно до документа “Програми та методики тестування”.

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ О.А. Павлов

“___” _____ 2019 р.

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ НА ОСНОВІ МОДЕЛІ АКТОРІВ

Програма та методика тестування

КПІ.ІП-5206.045490.03.51

“ПОГОДЖЕНО”

Керівник проекту:

_____ Г.В. Ісаченко

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ А.В. Ждан-Пушкін

Київ – 2019 року

ЗМІСТ

1 ОБ'ЄКТ ВИПРОБУВАНЬ 3

2 МЕТА ТЕСТУВАННЯ 4

3 МЕТОДИ ТЕСТУВАННЯ 5

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ 6

1 ОБ’ЄКТ ВИПРОБУВАНЬ

Об’єктом випробувань є комплекс для роботи з розробленої мовою програмування, а саме інтерпретатор мови програмування та інструмент створення та налаштування розподілених середовищ.

2 МЕТА ТЕСТУВАННЯ

Під час проведення тестування має відбутися первірка:

– Відповідності розробленого інтерпретатора описаним синтаксису та семантиці мови програмування

– Надійність роботи семантичного аналізу та перевірка типів

– Процес виконання мови програмування у створеному середовищі

– Підключення та налаштування розподіленого середовища у різних початкових умовах

3 МЕТОДИ ТЕСТУВАННЯ

Тестування проводилося за методом закритого продукту, тобто порівнянням результату та очікуваних даних без врахування особливостей архітектури продукту. Було проведене базове інтеграційне тестування на сумісну роботу інструменту налагодження середовищ та інтерпретатору, а також компонентне тестування для окремої перевірки кожного модулю.

4 ЗАСОБИ ТА ПОРЯДОК ТЕСТУВАННЯ

Тестування виконується за допомогою розробленого комплексу, налаштувань середовищ у форматі YAML, а також засобів контролю мереж операційної системи Linux, а саме iptables, netcat, netstat тощо.

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ О.А. Павлов

“___” _____ 2019 р.

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ НА ОСНОВІ МОДЕЛІ АКТОРІВ

Керівництво програміста

КПІ.ІП-5206.045490.04.33

“ПОГОДЖЕНО”

Керівник проекту:

_____ Г.В. Ісаченко

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____ А.В. Ждан-Пушкін

Київ – 2019 року

ЗМІСТ

1 ЗАГАЛЬНИЙ ОПИС МОВИ ПРОГРАМУВАННЯ FRISBEE 3

1.1 Система типів 3

1.2 Пасивні та активні об’єкти 3

1.3 Написання повноцінних програм 5

1.4 Розподілене середовище 6

2 ПРИКЛАД ПРОСТОЇ ПРОГРАМИ НА МОВІ FRISBEE 8

1 ЗАГАЛЬНИЙ ОПИС МОВИ ПРОГРАМУВАННЯ FRISBEE

Frisbee – імперативна мова програмування загального призначення, що базується на моделі акторів. За синтаксисом та способом виконання нагадує мови сімейства C, але також має концепції акторів (активних об’єктів) для використання можливостей паралельних обчислень.

1.1 Система типів

Мова має чотири основні типи даних та декілька способів їх розширення. Вбудованими способами розширення є, наприклад, масиви чи неповні типи – типи, які дозволяють містити пусте значення. Приклади оголошення таких типів наведено нижче.

```
String text = “Hello world”;  # Строковий тип #
Int number = 1898;  # Цілочисельний тип #
Bool isvalid = true;  # Логічний тип #

# Масив із строк #
[String] names = [“Anton”, “Zhdan”, “Pushkin”];

# Неповний цілочисельний тип – містить число чи void #
Int result = void;
```

Frisbee є статично типізованою мовою програмування, але надає можливість уникнути перевірки типів та мати динамічну типізацію за використання ключового слова `val`.

1.2 Пасивні та активні об’єкти

Користувач має можливість також вводити власні типи даних до мови, за допомогою оголошення нових пасивних або активних об’єктів. Пасивні об’єкти відповідають об’єктам у більшості мов програмування – вони мають певні поля, у яких зберігають значення, а також ряд методів, які можна викликати. Усі поля та методи у пасивних є публічними.

```
passive Person {
   String name;
   String surname;

   def Person create_relative(String new_name) {
      return new Person(new_name, this.surname);
   }
}

Person andy = new Person("Andy", "Warhol");
Person not_andy = andy.create_relative("Jack");
```

Вбудовані простіші типи даних також є пасивними об'єктами, які не мають полів, а лише методи.

```
[String] years = "2000,2004,2008,2012".split(",");
String age = 18.to_string()
```

Активні об'єкти є дуже схожими до пасивних за декількома виключеннями. Перш за все, активні об'єкти взаємодіють одне з одним за допомогою повідомлень і напряму не отримують результату обробки повідомлення. Кожен активних об'єкт виконується у окремому процесі операційної системи та має ізольовану пам'ять, саме тому повідомлення і є лише одним способом комунікації. Через ці самі причини поля активного об'єкту також недоступні. Лише сам активних об'єкт та пасивні об'єкти у його ізольованій пам'яті можуть використовувати його як пасивний об'єкт за допомогою ключового слова `this`.

Під час передачі повідомлень між активними об'єктами пасивні об'єкти цілком копіюються, а активні передаються лише як посилання на процес. Цей механізм може бути використаний для отримання відповіді.

```
active Greeter {
   def Void Person say_hello_to(Caller c) {
      # Приклад відправлення повідомлення #
      c ! hi("Hello from Greeter!");
   }
}
```

```
active Caller {
   def Void hi(String res) {
      io ! print(res);
   }
   def Void run() {
      Greeter g = spawn Greeter();
      g ! say_hello_to(this);
   }
}
```

Іншим способом очікувати на результат є використання механізму зворотніх повідомлень. Він дозволяє приховати деталі комунікації та автоматично відправляє позивачу відповідь повідомлення після завершення обробки повідомлення. Позивач в свою чергу у момент очікування фільтрує надходження і реагує лише на відповідь яку очікує, і лише після неї продовжує обробку інших повідомлень.

```
active Greeter {
   def Person say_hello() {
      return “Hello from Greeter!”;
   }
}

active Caller {
   def Void run() {
      Greeter g = spawn Greeter();

      # Приклад очікування на відповідь #
      String res <= g ! say_hello();
      io ! print(res);
   }
}
```

1.3 Написання повноцінних програм

Під час запуску програми написаної даної мовою програмування точкою входу є повідомлення run до активного об'єкту Main. Таким чином, саме цей клас є обов’язковим до оголошення для роботи програми.

```
active Main {
   def Void run() { io ! print(“Hello world!”);
}
```

Мова програмування має вбудований активний об'єкт `io`, який дозволяє виводити повідомлення на термінал. Для використання кращих методів вводу виводу рекомендується використовувати вбудовану бібліотеку `sockets`. Вбудовані бібліотеки, як і написані користувачем файли, можуть бути імпортовані за допомогою синтаксису підключення модулів.

```
from sockets import TCPServer;

active Main {
   def Void run() {
     TCPServer s = spawn TCPServer(9999); # Порт сокету #

     while (true) {
        val conn <= s ! accept();
        io ! print("New connection: ", conn);
     }
   }
}
```

1.4 Розподілене середовище

Основною перевагою розробленої мови програмування є можливість її використання у розподіленому середовищі. Для початку роботи з ним необхідно написати конфігураційний файл, а потім почергово запускати необхідні середовища та програми.

```
# Configuration.yaml #
master:
  ip: 142.93.236.104
  port: 7777

node1:
  ip: 142.93.128.161
  port: 7778
  connections:
    - master

node2:
  ip: 165.22.192.95
  port: 7777
  connections:
    - master
```

```
# master.frisbee #
active Main {
   val connections;

   def Void run() {
     io ! print("MASTER STARTED");
     this.connections = [];

     this ! connect(this);
   }

   def Void connect(val actor) {
     io ! print("NEW CONNECTION");
     this.connections = this.connections + [actor];
     io ! print(this.connections);
   }
}


# node.frisbee #
active Main {
   def Void run(val master) {
     io ! print("NODE STARTED");
     master ! connect(this);
   }
}


# Команди для запуску головної програми #
Python environ.py –c configuration.yaml master
Python main.py master.frisbee

# Команди для запуску допоміжних програм #
Python environ.py –c configuration.yaml node1
Python main.py node.frisbee
```

Під час запуску програми до її методу `Main.run` потрапляють посилання на процеси `Main.run` середовищ, які були вказані у конфігураційному файлі. Через це важливим є запускати головні середовища і програми перед запуском допоміжних.

2 ПРИКЛАД ПРОСТОЇ ПРОГРАМИ НА МОВІ FRISBEE

У даному розділі наведено приклад простої програми, яка відкриває сокет у системі, очікує введення даних та відправляє ці ж дані назад, але попередньо видаливши певну букву з повідомлення.

```
from sockets import LocalSocket;

passive StringLetterRemover {
   String initial;

   def String del(String letter) {
      [String] letters = this.initial.to_array()
      val result;

      Int i = 0;
      while (i < letters.length()) {
         if (letters[i] != letter)
            result = result + letters[i];
      }

      return result;
   }
}

active Main {
   def Void run() {
      Socket sock = spawn LocalSocket(10508);

      String text <= s ! get();
      io ! print("Got text:" + text);

      text = (new StringLetterRemover(text)).del("a");

      sock ! put("Hello, " + text);
   }
}
```

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ О.А. Павлов

“___” _____ 2019 р.

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ НА ОСНОВІ МОДЕЛІ АКТОРІВ

Опис програми

КПІ.ІП-5206.045490.05.13

“ПОГОДЖЕНО”

Керівник проекту:

_____ Г.В. Ісаченко

Нормоконтроль:

_____ К.І. Ліщук

Виконавець:

_____А.В. Ждан-Пушкін

Київ – 2019 року

Тексти програмного коду

Розробка мови програмування для розподілених обчислень на основі моделі акторів

DVD-R

(Вид носія даних)

17 арк, 95.8 Кб

(Обсяг програми (документа), арк.) Кб)

Київ – 2019

parser/src/ Tokens.x

```
{
module Tokens where
}

%wrapper "posn"

$digit = 0-9            -- digits
$alpha = [a-zA-Z]        -- alphabetic characters
$loweralpha = [a-z]
$upperalpha = [A-Z]
$graphic    = [$printable $white]

@string     = \" ($graphic # \")* \"
@comment     = \# ($graphic # \")* \#



tokens :-

  $white+                ;
  "active"                { \p s -> TActive p }
  "passive"                { \p s -> TPassive p }
  "new"                    { \p s -> TNew p }
  "spawn"                    { \p s -> TSpawn p }
  "import"                    { \p s -> TImport p }
  "from"                    { \p s -> TFrom p }

  "return"                                { \p s -> TReturn p }
  "def"                { \p s -> TDef p }

  "Void"                { \p s -> TVoid p }
  "val"                                { \p s -> TVal p }
  "String"                { \p s -> TString p }
  "Int"                    { \p s -> TInt p }
  "Bool"                { \p s -> TBool p }
  $upperalpha[$alpha $digit]*        { \p s -> TTypeIdent p s }
  "?"                    { \p s -> TMaybe p }

  "io"                                    { \p s -> TIo p }
  "if"                    { \p s -> TIf p }
  "else"                { \p s -> TElse p }
  "void"                { \p s -> TVoidValue p }
  "true"                { \p s -> TTrue p }
  "false"                { \p s -> TFalse p }
  "this"                { \p s -> TThis p }
  "while"                { \p s -> TWhile p }
  $digit+                { \p s -> TIntLiteral p (read s) }
  "."                                { \p s -> TPeriod p }

  "and"                    { \p s -> TOp p "and" }
  "not"                    { \p s -> TNot p }
  "or"                    { \p s -> TOp p "or" }

  [\+\-\*\/]                            { \p s -> TOp p [head s] }

  "<"                                { \p s -> TComOp p "<" }
  ">"                                { \p s -> TComOp p ">" }
  "=="                                { \p s -> TComOp p "==" }
```

```
  "!="                                  { \p s -> TComOp p "!=" }

  "="                    { \p s -> TEquals p }
  "<="                    { \p s -> TWaitMessage p }
  "!"                    { \p s -> TSendMessage p }

  ";"                     { \p s -> TSemiColon p }
  "("                    { \p s -> TLeftParen p }
  ")"                    { \p s -> TRightParen p }
  $loweralpha[$alpha $digit \_ \']*        { \p s -> TIdent p s }
  @string                            { \p s -> TStringLiteral p (init (tail s)) -- remove
the leading and trailing double quotes }
  @comment                                  { \p s -> TComment p s }
  "{"                          { \p s -> TLeftBrace p }
  "}"                    { \p s -> TRightBrace p }
  ","                    { \p s -> TComma p }
  "["                    { \p s -> TLeftBrack p }
  "]"                    { \p s -> TRightBrack p }
{
-- Each action has type ::AlexPosn -> String -> Token

-- The token type:
data Token =
         TLeftBrace AlexPosn        |
    TRightBrace AlexPosn            |
    TComma AlexPosn                 |
    TLeftBrack AlexPosn             |
    TRightBrack AlexPosn            |
    TActive AlexPosn                |
        TPassive AlexPosn           |
    TDef AlexPosn                   |
    TString AlexPosn                |
        TComment AlexPosn    String |
        TVal AlexPosn               |
    TVoid AlexPosn                  |
        TVoidValue AlexPosn         |
    TInt AlexPosn                   |
    TBool AlexPosn                  |
        TIo AlexPosn                |
    TIf AlexPosn                    |
    TElse AlexPosn                  |
    TTrue AlexPosn                  |
    TFalse AlexPosn                 |
    TThis AlexPosn                  |
    TWhile AlexPosn                 |
    TNew AlexPosn                   |
        TSpawn AlexPosn             |
        TImport AlexPosn            |
        TFrom AlexPosn              |
    TOp AlexPosn String             |
    TComOp AlexPosn String          |
        TMaybe AlexPosn             |
        TNot AlexPosn               |
    TEquals AlexPosn                |
        TWaitMessage AlexPosn       |
        TSendMessage AlexPosn       |
    TPeriod AlexPosn                |
    TSemiColon AlexPosn             |
    TLeftParen AlexPosn             |
    TRightParen AlexPosn            |
```

```
    TIdent AlexPosn String          |
        TTypeIdent AlexPosn String  |
    TIntLiteral AlexPosn Int        |
    TStringLiteral AlexPosn String  |
        TReturn AlexPosn
    deriving (Eq,Show)
```

parser/src/Frisbee.y

```
%name frisbee
%tokentype { Token }
%error { parseError }
%token
  "active"          { TActive _ }
  "passive"         { TPassive _ }
  "new"             { TNew _ }
  "spawn"           { TSpawn _ }
  "import"          { TImport _ }
  "from"            { TFrom _ }
  typeident         { TTypeIdent _ $$ }

  "Void"            { TVoid _ }
  "def"             { TDef _ }
  "return"          { TReturn _ }

  "val"             { TVal _ }
  "String"          { TString _ }
  "Int"             { TInt _ }
  "Bool"            { TBool _ }
  "?"               { TMaybe _ }
  "["               { TLeftBrack _ }
  "]"               { TRightBrack _ }
  "io"              { TIo _ }
  "if"              { TIf _ }
  "else"            { TElse _ }
  "void"            { TVoidValue _ }
  "true"            { TTrue _ }
  "false"           { TFalse _ }
  "this"            { TThis _ }
  "while"           { TWhile _ }
  integer_literal   { TIntLiteral _ $$ }
  string_literal    { TStringLiteral _ $$ }
  ident             { TIdent _ $$ }
  "{"               { TLeftBrace _ }
  "}"               { TRightBrace _ }
  ","               { TComma _ }

  op                { TOp _ $$}
  comop             { TComOp _ $$ }
  "("               { TLeftParen _ }
  ")"               { TRightParen _ }
  ";"               { TSemiColon _ }
  "."               { TPeriod _ }

  "not"             { TNot _ }
  "="               { TEquals _ }
  "<="              { TWaitMessage _ }
  "!"               { TSendMessage _ }

%left op
```

```
%nonassoc comop
%%

Program :
        ImportDeclList ObjectDeclList { Program $1 $2 }


ImportDeclList :
        "from" ident "import" ImportIdentList ";" ImportDeclList { ImportDeclList $2 $4
$6 }
        |                                                        { ImportDeclListEmpty }


ImportIdentList :
        typeident "," ImportIdentList { ImportIdentList $1 $3 }
        | typeident               { ImportIdentList $1 ImportIdentListEmpty }


ObjectDeclList :
          ObjectDecl     { ObjectDeclList $1 OEmpty }
          | ObjectDecl ObjectDeclList { ObjectDeclList $1 $2 }
          |             { OEmpty }

ObjectDecl :
            "active" typeident "{" VarDeclList MethodDeclList "}"     { ActiveDecl  $2
$4 $5 }
          | "passive" typeident "{" VarDeclList MethodDeclList "}"    { PassiveDecl $2
$4 $5 }


MethodDeclList :
     MethodDecl                   { MethodDeclList $1 MEmpty }
     | MethodDecl MethodDeclList  { MethodDeclList $1 $2 }
     |                            { MEmpty }

MethodDecl :
       "def" Type ident "(" FormalList ")" "{" StatementList "}" { MethodDecl $2 $3 $5
$8 }

VarDeclList :
     Type ident ";" { VarDeclList $1 $2 VEmpty }
     | Type ident ";" VarDeclList { VarDeclList $1 $2 $4 }
     |              { VEmpty }

FormalList :
     Type ident       { FormalList $1 $2 FEmpty }
     | Type ident "," FormalList { FormalList $1 $2 $4 }
     |                  { FEmpty }

Type :
     "val"       { TypeAnonymous }
     | CertainType { $1 }

CertainType :
     CertainType "?"    { TypeMaybe $1 }
     | "[" CertainType "]"    { TypeArray $2 }
     | "Void"       { TypeVoid }
     | "Int"    { TypeInt }
     | "String"    { TypeString }
     | "Bool"    { TypeBool }
```

```
      | typeident    { TypeIdent $1 }


Statement :
    "{" StatementList "}"                              { SList $2 }
    | "if" "(" Exp ")" Statement "else" Statement  { SIfElse $3 $5 $7 }
    | "if" "(" Exp ")" Statement                 { SIfElse $3 $5 (SList Empty) }
    | "while" "(" Exp ")" Statement               { SWhile $3 $5 }
    | "return" Exp ";"                             { SReturn $2 }
    | ident "=" Exp ";"                            { SEqual $1 $3 }
    | Type ident ";"                              { SVarDecl $1 $2 }
    | Type ident "=" Exp ";"                              { SVarDeclEqual $1 $2 $4 }
    | Exp "." ident   "=" Exp ";"                              { SEqualField $1 $3 $5 }
    | Exp "!" ident "(" ExpList ")" ";"   { SSendMessage $1 $3 $5}
    | ident "<=" Exp "!" ident "(" ExpList ")" ";"   { SWaitMessage $1 $3 $5 $7 }
    | Type ident "<=" Exp "!" ident "(" ExpList ")" ";"   { SList (StatementList
                                                                  (StatementList
                                                                     Empty
                                                                     (SWaitMessage
$2 $4 $6 $8))
                                                                  (SVarDecl  $1  $2))
}
    | ident "[" Exp "]" "=" Exp ";"                  { SArrayEqual $1 $3 $6 }
    | Exp   ";"                     { SExp $1}

StatementList :
    Statement               { StatementList Empty $1 }
    | StatementList Statement   { StatementList $1 $2 }

Exp :
    Exp op Exp                        { ExpOp $1 $2 $3}
    | Exp comop Exp                   { ExpComOp $1 $2 $3}
    | Exp "[" Exp "]"                 { ExpArrayGet $1 $3}
    | "[" ExpList "]"                 { ExpArrayValue $2 }
    | Exp "." ident "(" ExpList ")"   { ExpFCall $1 $3 $5}
    | Exp "." ident                   { ExpFieldAccess $1 $3}
    | integer_literal                 { ExpInt $1}
    | string_literal                 { ExpString $1}
    | "void"                          { ExpVoid }
    | "true"                          { ExpBool True}
    | "false"                         { ExpBool False}
    | ident                           { ExpIdent $1}
    | "this"                          { ExpThis }
    | "io"                           { ExpIO }
    | "new" typeident "(" ExpList")"             { ExpNewPassive $2 $4}
    | "spawn" typeident "(" ExpList ")"             { ExpSpawnActive $2 $4}
    | "not" Exp                         { ExpNot $2}
    | "(" Exp ")"                       { ExpExp $2}

ExpList :
        Exp "," ExpList  { ExpList $1 $3 }
        | Exp            { ExpList $1 ExpListEmpty }
        | Exp  ","       { ExpList $1 ExpListEmpty }
        |                { ExpListEmpty }


data Program = Program ImportDeclList ObjectDeclList  -- imports, objects
      deriving (Show, Eq)
```

```
data ImportDeclList
    = ImportDeclList String ImportIdentList ImportDeclList  -- module, typenames, tail
    | ImportDeclListEmpty  --
    deriving (Show, Eq)


data ObjectDeclList
    = ObjectDeclList ObjectDecl ObjectDeclList  -- head, tail
    | OEmpty --
  deriving (Show, Eq)

data ObjectDecl
    = ActiveDecl  String VarDeclList MethodDeclList  -- name, vars, methods
    | PassiveDecl String VarDeclList MethodDeclList  -- name, vars, methods
  deriving (Show, Eq)


data MethodDeclList
    = MethodDeclList MethodDecl MethodDeclList  -- head, tail
    | MEmpty  --
    deriving (Show, Eq)


data MethodDecl
    = MethodDecl Type String FormalList StatementList  -- type, name, args, statements
    deriving (Show, Eq)

data VarDeclList =
    VarDeclList Type String VarDeclList  -- typename, name, tail
    | VEmpty  --
    deriving (Show, Eq)

data FormalList =
    FormalList Type String FormalList  -- typename, name, tail
    | FEmpty  --
  deriving (Show, Eq)

data Type =
      TypeAnonymous  --
    | TypeMaybe Type  -- type
    | TypeArray Type  -- type
    | TypeInt  --
    | TypeVoid  --
    | TypeBool  --
    | TypeString  --
    | TypeIdent String  -- name
    deriving (Show, Eq)

data Statement
    = SList StatementList  -- statements
    | SIfElse Exp Statement Statement  -- condition, if_body, else_body
    | SWhile Exp Statement  -- condition, body
    | SReturn Exp  -- expr
    | SEqual String Exp  -- name, expr
    | SVarDeclEqual Type String Exp  -- type, name, expr
    | SVarDecl Type String  -- type, name
    | SEqualField Exp String Exp  -- object, field, expr
    | SArrayEqual String Exp Exp  -- name, index, expr
    | SSendMessage Exp String ExpList  -- object, method, args
    | SWaitMessage String Exp String ExpList  -- result_name, object, method, args
```

```
    | SExp Exp  -- expr
    deriving (Show, Eq)

data StatementList
    = StatementList StatementList Statement -- tail, head
    | Empty --
    deriving (Show, Eq)


data Exp
    = ExpOp Exp String Exp  -- left, operator, right
    | ExpComOp Exp String Exp  -- left, operator, right
    | ExpArrayGet Exp Exp -- array, index
    | ExpArrayValue ExpList -- elements
    | ExpFCall Exp String ExpList  -- object, method, args
    | ExpFieldAccess Exp String  -- object, field
    | ExpInt Int  -- value
    | ExpString String  -- value
    | ExpBool Bool -- value
    | ExpVoid -- value
    | ExpIdent String -- name
    | ExpNewPassive String ExpList  -- typename, args
    | ExpSpawnActive String ExpList  -- typename, args
    | ExpExp Exp -- expr
    | ExpThis  --
    | ExpIO  --
    | ExpNot Exp  -- operand
    deriving (Show, Eq)

data ExpList
    = ExpList Exp ExpList  -- head, tail
    | ExpListEmpty    --
    deriving (Show, Eq)

data ImportIdentList
    = ImportIdentList String ImportIdentList  -- typename, tail
    | ImportIdentListEmpty   --
    deriving (Show, Eq)
}
```

evaluation/ast_def/statements.py

```
from __future__ import annotations

from dataclasses import dataclass

from .. import global_conf
from .expressions import BaseExp, ExpBool, ExpArray, BaseExpList, ExpInt, ActiveProxy
from .types import BaseType


@dataclass
class BaseStatement:
    def run(self, ctx) -> None:
        pass


@dataclass
class SList(BaseStatement):
    statements: BaseStatementList
```

```
    def run(self, ctx):
        self.statements.run(ctx)


@dataclass
class SIfElse(BaseStatement):
    condition: BaseExp
    if_body: BaseStatement
    else_body: BaseStatement

    def run(self, ctx):
        res: ExpBool = self.condition.evaluate(ctx)
        if res.value:
            self.if_body.run(ctx)
        else:
            self.else_body.run(ctx)


@dataclass
class SWhile(BaseStatement):
    condition: BaseExp
    body: BaseStatement

    def run(self, ctx):
        while self.condition.evaluate(ctx).value:
            self.body.run(ctx)

        return ctx


@dataclass
class SReturn(BaseStatement):
    expr: BaseExp

    def run(self, ctx):
        ctx['return'] = self.expr.evaluate(ctx)


@dataclass
class SEqual(BaseStatement):
    name: str
    expr: BaseExp

    def run(self, ctx):
        ctx['env'][self.name] = self.expr.evaluate(ctx)


@dataclass
class SEqualField(BaseStatement):
    object: BaseExp
    field: str
    expr: BaseExp

    def run(self, ctx):
        object = self.object.evaluate(ctx)
        object.set_field(self.field, self.expr.evaluate(ctx))


@dataclass
```

```
class SVarDeclEqual(BaseStatement):
    type: BaseType
    name: str
    expr: BaseExp

    def run(self, ctx):
        ctx['env'][self.name] = self.expr.evaluate(ctx)


@dataclass
class SVarDecl(BaseStatement):
    type: BaseType
    name: str

    def run(self, ctx):
        pass


@dataclass
class SArrayEqual(BaseStatement):
    name: str
    index: BaseExp
    expr: BaseExp

    def run(self, ctx):
        value = self.expr.evaluate(ctx)
        index = self.index.evaluate(ctx)

        assert isinstance(ctx['env'][self.name], ExpArray), "Not array!"
        assert isinstance(index, ExpInt), "Not int!"

        ctx['env'][self.name].array[index.value] = value


@dataclass
class SSendMessage(BaseStatement):
    object: BaseExp
    method: str
    args: BaseExpList

    def run(self, ctx):
        from ..active_object import ExpActiveObject
        object = self.object.evaluate(ctx)
        args = self.args.get_exprs(ctx)

        for i, o in enumerate(args):
            if isinstance(o, ExpActiveObject):
                args[i] = ActiveProxy(actor_id=o.actor_id)

        if isinstance(object, ExpActiveObject):
            object = ActiveProxy(actor_id=object.actor_id)

        object.send_message(self.method, args, return_to=None)


@dataclass
class SWaitMessage(BaseStatement):
    result_name: str
    object: BaseExp
    method: str
```

```
    args: BaseExpList

    def run(self, ctx):
        object: ActiveProxy = self.object.evaluate(ctx)
        object.send_message(
            self.method,
            self.args.get_exprs(ctx),
            return_to=global_conf.local_connector.actor_id
        )

        ctx['env'][self.result_name] =
global_conf.local_connector.receive_return_value()


@dataclass
class SExp(BaseStatement):
    expr: BaseExp

    def run(self, ctx):
        self.expr.evaluate(ctx)
```

evaluation/active_object.py

```
from __future__ import annotations

import multiprocessing as mp
import typing
from dataclasses import dataclass, field

from . import global_conf
from .connector import ActorConnector
from .ast_def.declarations import BaseObjectDecl, BaseMethodDeclList, BaseVarDeclList,
MethodDecl
from .ast_def.expressions import BaseExp, ActiveProxy


class BaseActiveObjectDeclaration(BaseObjectDecl):

    def spawn(self, args: typing.List[BaseExp]) -> ActiveProxy:
        return NotImplemented


@dataclass
class ActiveDecl(BaseActiveObjectDeclaration):
    name: str
    vars: BaseVarDeclList
    methods: BaseMethodDeclList

    module: str = field(default_factory=lambda: 'NOT_FOUND')

    def get_methods(self):
        methods = self.methods.get_methods()
        return {m.name: m for m in methods}

    def spawn(self, args):
        field_names = [name for name, type in self.vars.get_fields()]
        fields = dict(zip(field_names, args))

        new_active = ExpActiveObject(env=fields, module=self.module,
typename=self.name)
```

```
        return new_active.start_and_return_proxy()


class BaseActiveObject:
    actor_id: str

    @staticmethod
    def _actor_loop(actor_obj: BaseActiveObject, event: mp.Event, assigned_id:
mp.Array):
        global_conf.local_connector = ActorConnector()

        assigned_id.value = global_conf.local_connector.actor_id.encode('ascii')
        actor_obj.actor_id = global_conf.local_connector.actor_id

        event.set()

        actor_obj.on_start()
        while True:
            message_name, args, return_address =
global_conf.local_connector.receive_message()

            result = actor_obj.proceed_message(message_name, args)
            if return_address:
                global_conf.local_connector.return_result(return_address, result)

    def start_and_return_proxy(self) -> ActiveProxy:
        spawned_event = mp.Event()
        assigned_id = mp.Array('c', 64)

        proc = mp.Process(target=self._actor_loop, args=(self, spawned_event,
assigned_id))
        proc.start()
        spawned_event.wait()

        return ActiveProxy(actor_id=assigned_id.value.decode('ascii'))

    def on_start(self):
        pass

    def proceed_message(self, message_name: str, args: typing.List[BaseExp]) ->
BaseExp:
        return NotImplemented


@dataclass
class ExpActiveObject(BaseActiveObject):
    env: typing.Dict[str, BaseExp]
    module: str
    typename: str

    @property
    def declaration(self):
        return global_conf.types_mapping[self.module][self.typename]

    def get_field(self, name):
        return self.env[name]

    def set_field(self, name, value):
        self.env[name] = value
```

```
    def proceed_message(self, name, args):
        method: MethodDecl = self.declaration.get_methods()[name]
        return method.execute(this=self, args=args)

    def run_method(self, name, args):
        method: MethodDecl = self.declaration.get_methods()[name]
        return method.execute(this=self, args=args)
```

evaluation/connector.py

```
import uuid
import zmq
import time

from .ast_def.expressions import *
from .passive_object import ExpPassiveObject


class ActorConnector:
    actor_id: str

    messages_socket: zmq.Socket
    return_socket: zmq.Socket
    write_socket: zmq.Socket

    def __init__(self, ):
        self.actor_id = str(uuid.uuid4())

        context = zmq.Context()
        self.messages_socket = context.socket(zmq.SUB)
        self.messages_socket.connect('tcp://127.0.0.1:5556')
        self.messages_socket.subscribe(f'messages:{self.actor_id}')
        # messages_socket.subscribe('')

        self.return_socket = context.socket(zmq.SUB)
        self.return_socket.connect('tcp://127.0.0.1:5556')
        self.return_socket.subscribe(f'return:{self.actor_id}')

        self.write_socket = context.socket(zmq.PUB)
        self.write_socket.connect('tcp://127.0.0.1:5557')

        time.sleep(0.2)  # ensure connection established

    def receive_message(self):
        topic, data = self.messages_socket.recv_multipart()
        data = eval(data)
        return data['name'], data['args'], data['return']

    def receive_return_value(self):
        topic, result = self.return_socket.recv_multipart()
        return eval(result.decode('ascii'))

    def return_result(self, return_actor, result):
        self.write_socket.send_multipart([
            'return:{}'.format(return_actor).encode('ascii'),
            str(result).encode('ascii')
        ])

    def send_message(self, actor_id, name, args, return_to: str = None):
        self.write_socket.send_multipart([
```

```
            actor_id.encode('ascii'),
            str({'name': name, 'args': args, 'return': return_to}).encode('ascii')
        ])


def send_initial_message(actor_id, name, args):
    context = zmq.Context()

    write_socket = context.socket(zmq.PUB)
    write_socket.connect('tcp://127.0.0.1:5557')
    time.sleep(0.2)
    write_socket.send_multipart([
        actor_id.encode('ascii'),
        str({'name': name, 'args': args, 'return': None}).encode('ascii')
    ])
```

evaluation/global_conf.py

```
import typing

local_connector: typing.Any  # 'connector.ActorConnector'
types_mapping: typing.Dict[str, typing.Dict[str, typing.Any]]
```

environ.py

```
import socket
import sys
import yaml
import zmq
import threading
import uuid


ENV_UUID = str(uuid.uuid4())

config = yaml.safe_load(open(sys.argv[1]))
local_conf = config[sys.argv[2]]

global_read_port = None
global_write_port = None

main_actor: str
local_actors = []

connected = {

}

def start_zmq_queues():
    global global_read_port
    global global_write_port
    global main_actor
    global local_actors

    c = zmq.Context()

    read = c.socket(zmq.SUB)
    global_read_port = read.bind_to_random_port('tcp://127.0.0.1')
    read.subscribe('')  # Read all topics
```

```
    write = c.socket(zmq.PUB)
    global_write_port = write.bind_to_random_port('tcp://127.0.0.1')

    print(
        'Starting read queue on ',
        global_read_port,
        'and write queue on',
        global_write_port
    )

    while True:
        topic, data = read.recv_multipart()
        print(topic)
        typeof, actor_id = topic.decode('ascii').split(':')
        print('[{:7}][{}] {}'.format(typeof, actor_id, data))

        if typeof == 'return':
            res_topic = f'return:{actor_id}'
        elif typeof == 'message':
            res_topic = f'messages:{actor_id}'
        elif typeof == 'main':
            main_actor = actor_id
            continue
        elif typeof == 'create':
            local_actors.append(actor_id)
            continue
        else:
            continue

        write.send_multipart([res_topic.encode('ascii'), data])


def start_client(sock):
    while True:
        data = sock.recv(1024)
        if not data:
            sock.close()
            return

        data = data.decode('ascii')
        if data == 'init':
            print('#### NEW PROGRAM CONNECTING ####')
            sock.send(f'{global_write_port}:{global_read_port}'.encode('ascii'))
            sock.close()
            return
        elif data.startswith('remote:'):
            new_env_uuid = data.split(':')[1]
            print('NEW ENV UUID', new_env_uuid)
            sock.send(main_actor.encode('ascii'))
            sock.close()
            return


def start_tcp_server():
    print('Starting TCP server on', local_conf['port'])
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    sock.bind(('0.0.0.0', local_conf['port']))
    sock.listen()
```

```
    while True:
        client, _ = sock.accept()
        handler = threading.Thread(target=start_client, args=[client, ])
        handler.start()



if __name__ == '__main__':
    print('LAUNCHING ENV', ENV_UUID)
    connections = local_conf.get('connections', [])
    for c in connections:
        con_conf = config[c]
        print(con_conf)
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.connect((con_conf['ip'], int(con_conf['port'])))
        s.send(f'remote:{ENV_UUID}'.encode('ascii'))

        data = s.recv(1024).decode('ascii')
        main_uuid = data

    threading.Thread(target=start_zmq_queues).start()

    start_tcp_server()
```

Факультет інформатики та обчислювальної техніки

Кафедра автоматизованих систем обробки інформації і управління

“ЗАТВЕРДЖЕНО”

В.о. завідувача кафедри

_____ О.А. Павлов

“___” _____ 2019 р.

РОЗРОБКА МОВИ ПРОГРАМУВАННЯ ДЛЯ РОЗПОДІЛЕНИХ ОБЧИСЛЕНЬ НА ОСНОВІ МОДЕЛІ АКТОРІВ

Графічні матеріали

КПІ.ІП-5206.045490.06.99

“ПОГОДЖЕНО”

Керівник проекту:

_____ Г.В. Ісаченко

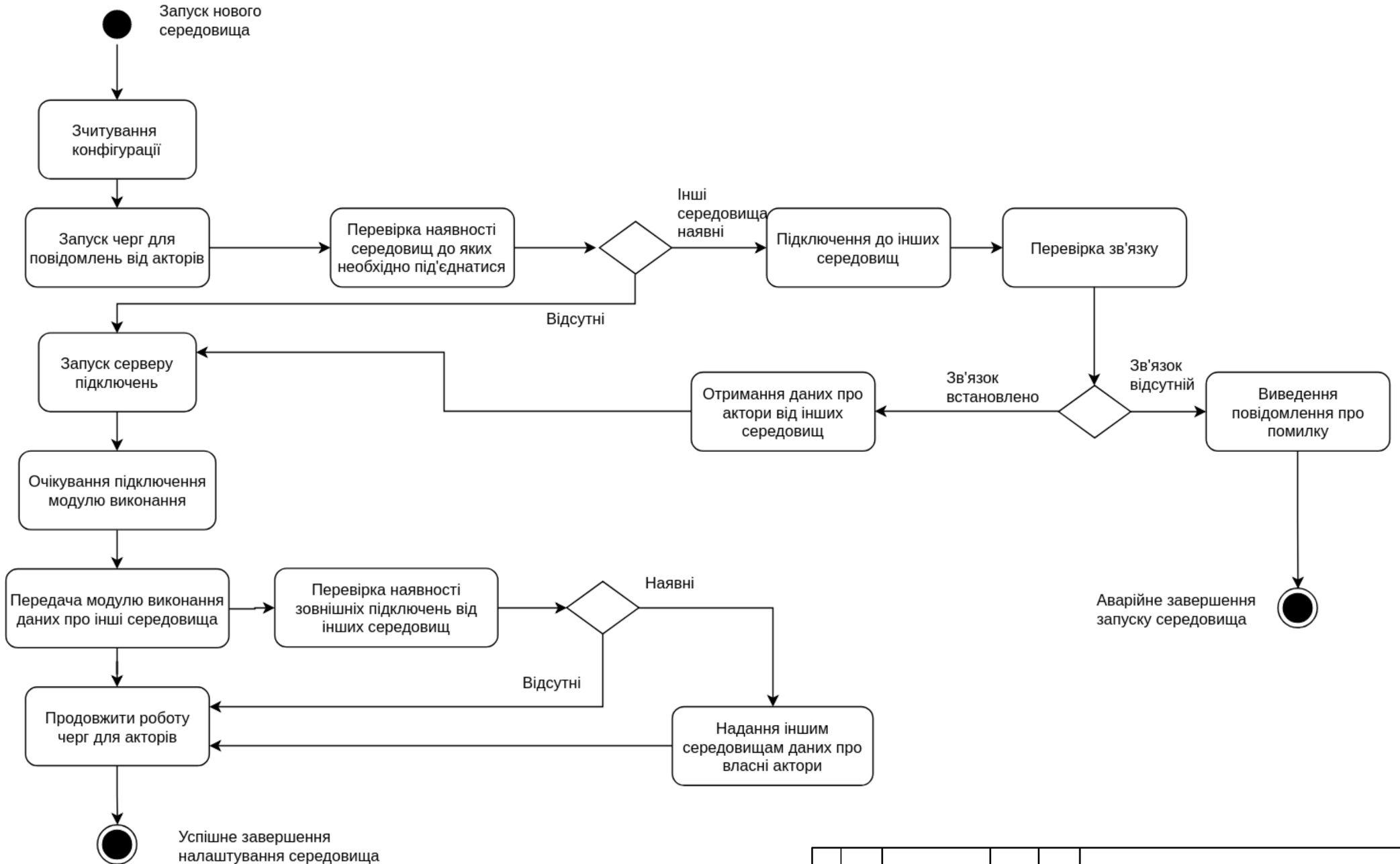
Нормоконтроль:

_____ К.І. Ліщук

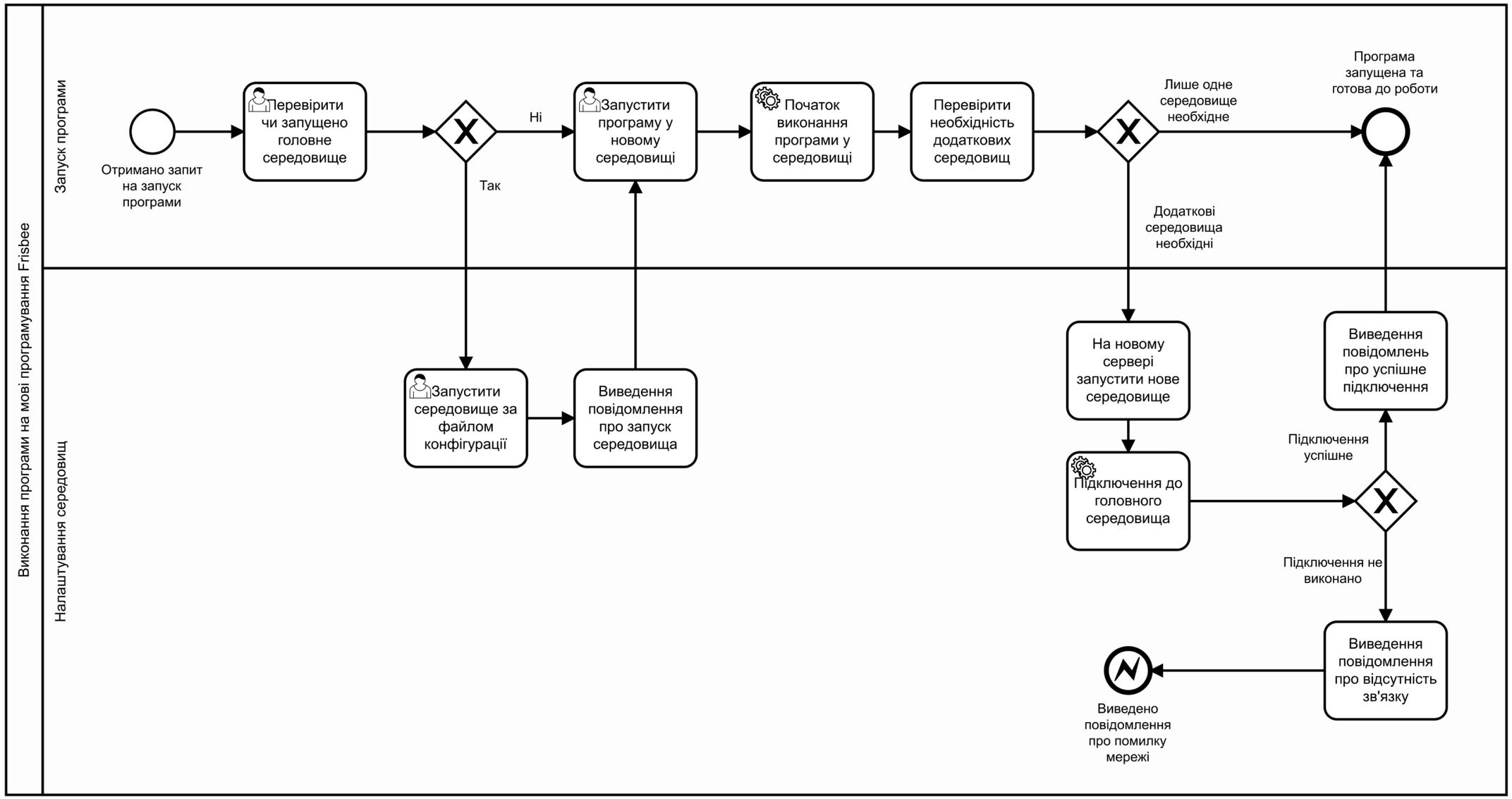
Виконавець:

_____А.В. Ждан-Пушкін

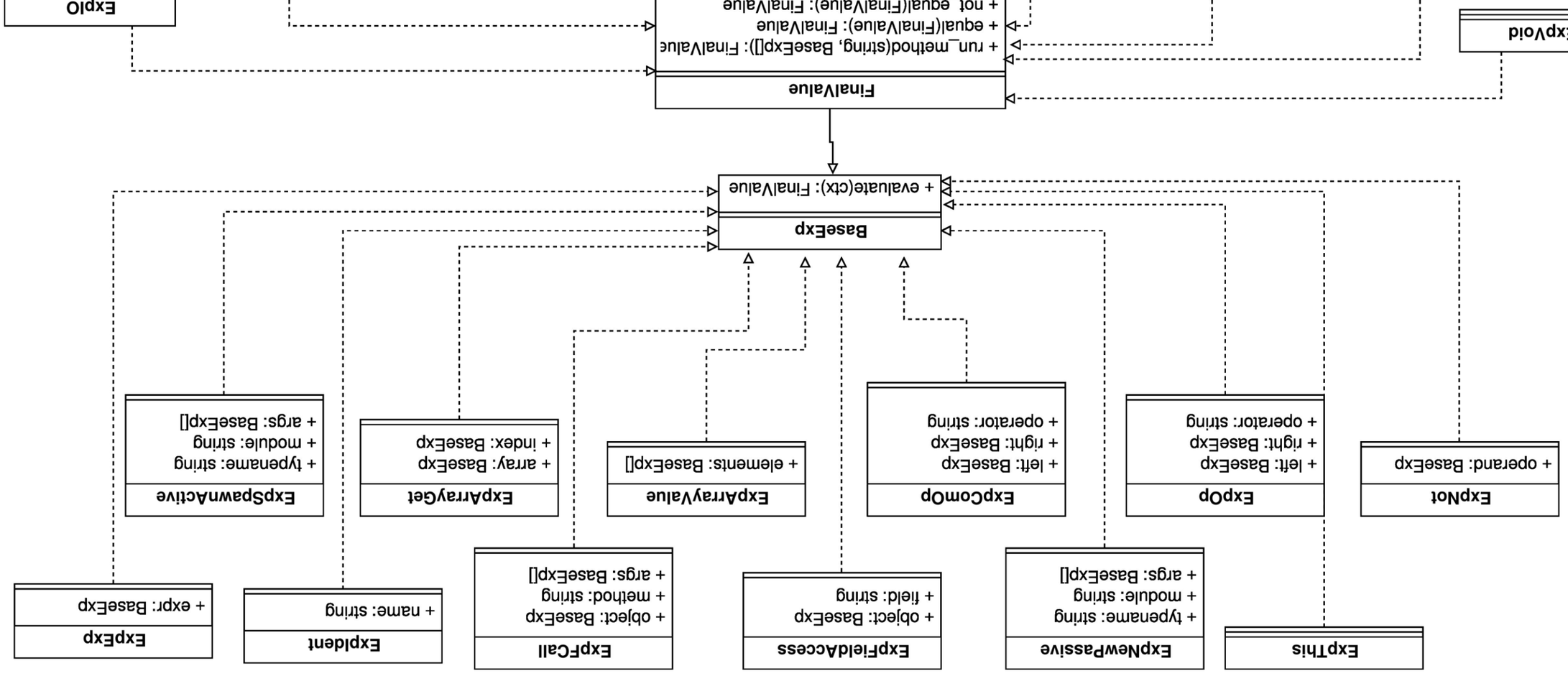
Київ – 2019 року



					КПІ.ІП-5206.045490.06.99 СС	
						Літера / Маса / Масштаб
Зм.	Арк.	№ документа	Підпис	Дата	*Схема структурна діяльності роботи програмного забезпечення*	
Розробив		Ждан-Пушкін А.В				Аркуш / Аркушів
Перевірив		Ісаченко Г.В.				
Т. кон.						
Н. кон.		Ліщук К.І.			Розробка мови програмування для розподілених обчислень на основі моделі акторів	КПІ ім. Ігоря Сікорського Кафедра АСОІУ гр. ІП-52
Затвердив		Ісаченко Г.В.				



					КПІ.ІП-5206.045490.06.99 СБП		
						Літера	Маса / Масштаб
Зм.	Арк.	№ документа	Підпис	Дата	*Схема бізнес-процесу*		
Розробив		Ждан-Пушкін А.В				Аркуш	Аркушів
Перевірив		Ісаченко Г.В.					
Т. кон.							
					Розробка мови програмування для розподілених обчислень на основі моделі акторів	КПІ ім. Ігоря Сікорського Кафедра АСОІУ гр. ІП-52	
Н. кон.		Ліщук К.І.					
Затвердив		Ісаченко Г.В.					



КПІ.ІП-5206.045490.06.99 СС

									КПІ.ІП-5206.045490.06.99 СС		
Зм.	Арк.	№ документа	Підпис	Дата							
Розробив		Жданн-Глушкін А.В.			Схема структурна класів програмного забезпечення				Літера	Маса	Масштаб
Перевірив		Іванченко Г.В.									
Т. контр.									Аркуш	Аркушів	
					Розробка мови програмування для розподілених обчислень на основі моделі акторів				КПІ ім. Ігоря Сікорського Кафедра АСОІУ гр. ІП-52		
Н. контр.		Глущк К.І.									
Затвердив		Іванченко Г.В.									